hankel Documentation

Release 0.3.5

Steven Murray

Dec 14, 2017

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

Quicklinks

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

CHAPTER 2

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 (as of v0.2.0).

CHAPTER 3

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

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, by looking at some of the tests, or by looking at whatever is in the `devel/` directory at the time.

So, what would you like to learn?

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:

```
In [1]: from hankel import HankelTransform      # Import the basic class

        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:

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

```
Out[3]: (1.0000000000003486, -9.8381428368537518e-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):

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

```
Out[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:

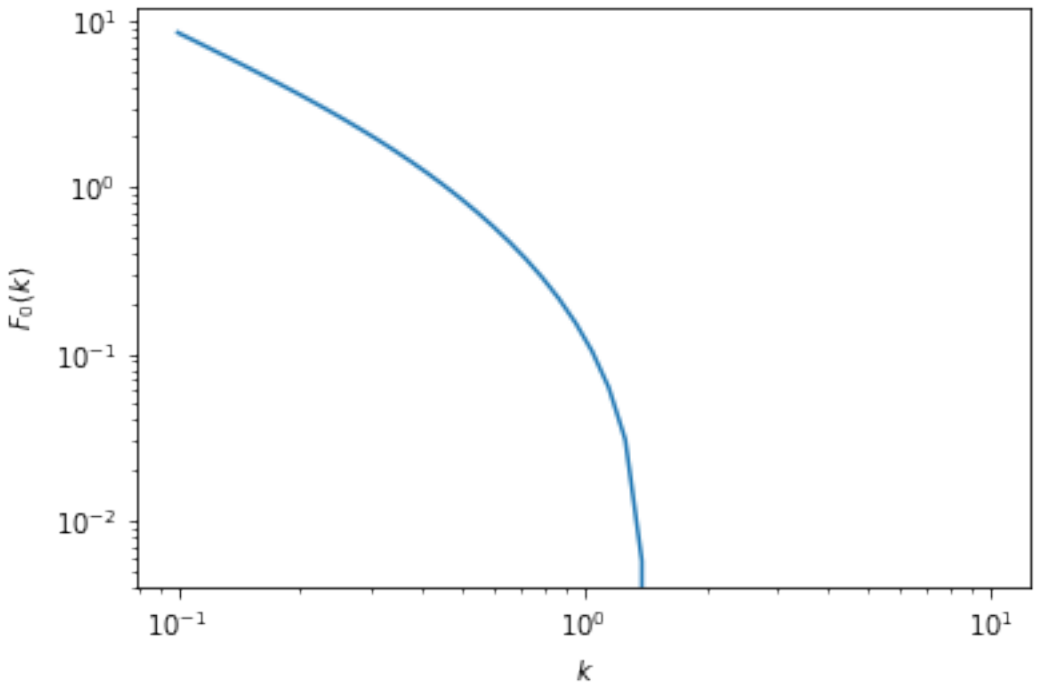
```
In [7]: import numpy as np               # Import numpy
        import matplotlib.pyplot as plt
        %matplotlib inline
```

Now do the transform,

```
In [ ]: 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:

```
In [11]: plt.plot(k,Fk)
         plt.xscale('log')
         plt.yscale('log')
         plt.ylabel(r"$F_0(k)$")
         plt.xlabel(r"$k$")
         plt.show()
```



Fourier Transforms

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

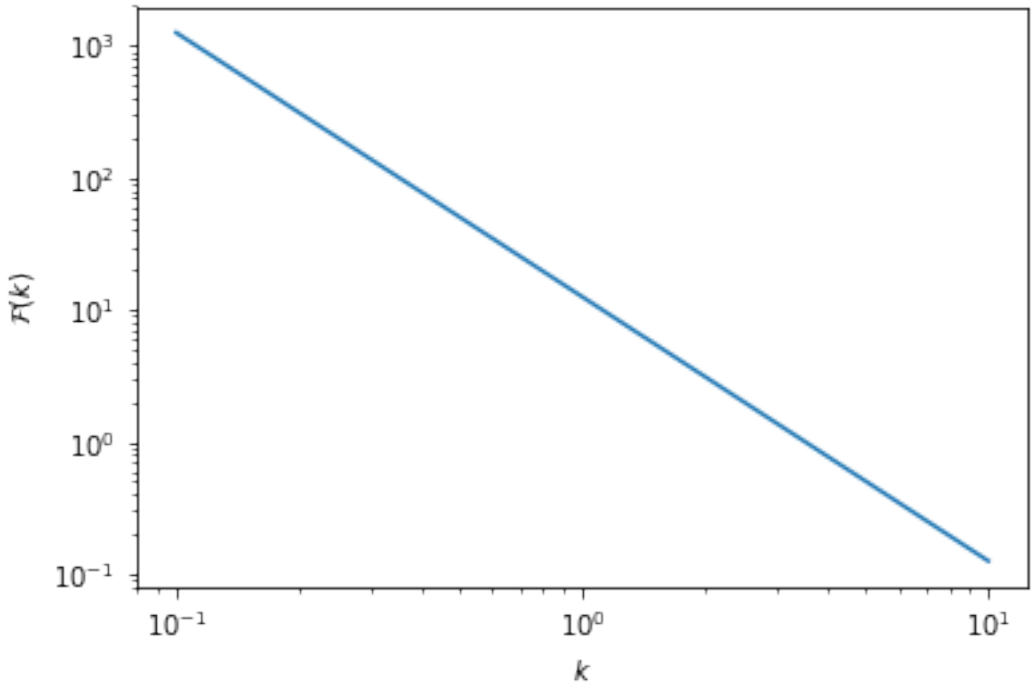
$$F(k) = \frac{(2\pi)^{n/2}}{k^{n/2-1}} \int_0^\infty r^{n/2-1} f(r) J_{n/2-1}(kr) r dr.$$

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:

```
In [12]: # 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)
```

```
In [13]: 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

In terms of limitations of the method, they are very dependent on the form of the function chosen. Notably, functions which tend to infinity at x=0 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:

```
In [14]: f = lambda x: 1/np.sqrt(x)
         h = HankelTransform(0.5,120,0.03)
         h.integrate(f)
```

```
Out[14]: (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:

```
In [15]: h = HankelTransform(0.5,700,0.001)
         h.integrate(f)
```

```
Out[15]: (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. This limitation is amplified for cases where the function really does tend to infinity at $x = 0$, rather than a finite positive number, such as $f(x) = 1/x$. Clearly the integral becomes non-convergent for some $f(x)$, in which case the numerical approximation can never be correct.

Upper-Bound Convergence

If the function $f(x)$ is monotonically increasing, or at least very slowly decreasing, then 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 $f(x) = x^{0.5}$ and no more (otherwise they diverge). 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:

```
In [16]: h = HankelTransform(0.5,700,0.001)
         f = lambda x : x**0.4
         h.integrate(f)
```

```
Out[16]: (0.5367827792529053, -1.0590954621251101)
```

The analytic result is 0.8421449 – very far from our result. Note that in this case, the error estimate itself is a good indication that we haven't reached convergence. We could try increasing `N`:

```
In [17]: h = HankelTransform(0.5,10000,0.001)
         h.integrate(f,ret_err=False)/0.8421449 -1
```

```
hankel/hankel.py:72: RuntimeWarning: overflow encountered in sinh
  a = (np.pi*t*np.cosh(t) + np.sinh(np.pi*np.sinh(t)))/(1.0 + np.cosh(np.pi*np.sinh(t)))
hankel/hankel.py:72: RuntimeWarning: overflow encountered in cosh
  a = (np.pi*t*np.cosh(t) + np.sinh(np.pi*np.sinh(t)))/(1.0 + np.cosh(np.pi*np.sinh(t)))
hankel/hankel.py:72: RuntimeWarning: invalid value encountered in divide
  a = (np.pi*t*np.cosh(t) + np.sinh(np.pi*np.sinh(t)))/(1.0 + np.cosh(np.pi*np.sinh(t)))
```

```
Out[17]: 7.1335646079084825e-07
```

This is very accurate, but quite slow. Alternatively, we could try increasing `h`:

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

```
Out[18]: 0.00045613842025526985
```

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

There are some notebooks in the `devel/` directory which toy with some known integrals, and show how accurate different choices of `N` and `h` are. They are interesting to view to see some of the patterns.

5.1.2 Compare Hankel and Fourier Transforms

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,

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

$$In this notebook we will consider two problems : 1. Given : math : `f(r)`,$$

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

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

$$2. Differentiate in Spectral space$$

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

$$3. Compute the Inverse Transform$$

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

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.

Import Relevant Libraries

```
In [2]: # Import Libraries

        import numpy as np                                                    # Numpy
        from scipy.fftpack import fft2, ifft2, fftfreq, ifftn, fftn           # Fourier
        from hankel import HankelTransform, SymmetricFourierTransform         # Hankel
        from scipy.interpolate import InterpolatedUnivariateSpline as spline  # Splines
        import matplotlib.pyplot as plt                                       # Plotting
        import matplotlib as mpl
        from os import path
        %matplotlib inline
```

```
In [34]: ## Put the prefix to the figure directory here for your computer. If you don't want to save
         prefix =  path.expanduser("~/Documents/Projects/HANKEL/laplacian_paper/Figures/")
```

Standard Plot Aesthetics

```
In [4]: mpl.rcParams['lines.linewidth'] = 2
        mpl.rcParams['xtick.labelsize'] = 13
        mpl.rcParams['ytick.labelsize'] = 13
        mpl.rcParams['font.size'] = 15
        mpl.rcParams['axes.titlesize'] = 14
```

Define Sample Functions

We define the two functions

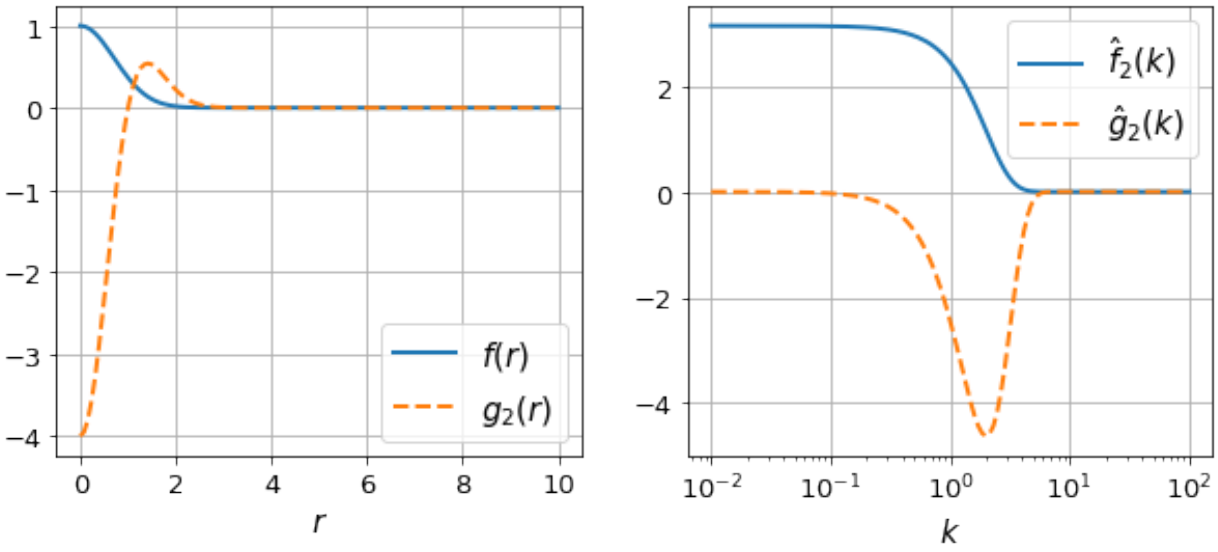
$$f = e^{-r^2} \quad \text{and} \quad g = 4e^{-r^2}(r^2 - 1).$$

$$It is easy to verify that they are related by the Laplacian operator.$$

```
In [5]: # Define Gaussian

        f = lambda r: np.exp(-r**2)

        # Define Laplacian Gaussian function

        g = lambda r: 4.0*np.exp(-r**2)*(r**2 - 1.0)
```

We can also define the FTs of these functions analytically, so we can compare our numerical results:

```
In [6]: fhat = lambda x : np.pi*np.exp(-x**2/4.)
        ghat = lambda x : -x**2*fhat(x)

In [7]: # Make a plot of the sample functions

        fig, ax = plt.subplots(1,2,figsize=(10,4))
        r = np.linspace(0,10,128)
        ax[0].plot(r, f(r), label=r"$f(r)$")
        ax[0].plot(r, g(r), label=r'$g_2(r)$', ls="--")
        ax[0].legend()
        ax[0].set_xlabel(r"$r$")
        ax[0].grid(True)

        k = np.logspace(-2,2,128)
        ax[1].plot(k, fhat(k), label=r"$\hat{f}_2(k)$")
        ax[1].plot(k, ghat(k), label=r'$\hat{g}_2(k)$', ls="--")
        ax[1].legend()
        ax[1].set_xlabel(r"$k$")
        ax[1].grid(True)
        ax[1].set_xscale('log')
        #plt.suptitle("Plot of Sample Functions")


        if prefix:
            plt.savefig(path.join(prefix,"sample_function.pdf"))
```



Define Transformation Functions

```
In [8]: def ft_transformation_2d(f,x, inverse=False):

            xx,yy = np.meshgrid(x,x)
            r = np.sqrt(xx**2 + yy**2)

            # Appropriate k-space values
            k = 2*np.pi*fftfreq(len(x),d=x[1]-x[0])
            kx,ky = np.meshgrid(k,k)
            K2 = kx**2+ky**2

            # The transformation
            if not inverse:
                g2d = ifft2(-K2 * fft2(f(r)).real).real
            else:
                invK2 = 1./K2
                invK2[np.isinf(invK2)] = 0.0

                g2d = ifft2(-invK2 * fft2(f(r)).real).real

            return x[len(x)//2:], g2d[len(x)//2,len(x)//2:]
```

```
In [27]: def ht_transformation_nd(f,N_forward,h_forward,K,r,ndim=2, inverse=False, N_back=None, h_bac
                                  ret_everything=False):

             if N_back is None:
                 N_back = N_forward
             if h_back is None:
                 h_back = h_forward

             # Get transform of f
             ht = SymmetricFourierTransform(ndim=ndim, N=N_forward, h=h_forward)

             if ret_everything:
                 fhat, fhat_cumsum = ht.transform(f, K, ret_cumsum=True, ret_err=False)
             else:
                 fhat = ht.transform(f, K, ret_err = False)

             # Spectral derivative
             if not inverse:
                 ghat = -K**2 * fhat
             else:
                 ghat = -1./K**2 * fhat

             # Transform back to physical space via spline
             # The following should give best resulting splines for most kinds of functions
             # Use log-space y if ghat is either all negative or all positive, otherwise linear-space
             # Use order 1 because if we have to extrapolate, this is more stable.
             # This will not be a good approximation for discontinuous functions... but they shouldn
             if np.all(ghat<=1e-13):
                 g_ = spline(K[ghat<0],np.log(-ghat[ghat<0]),k=1)
                 ghat_spline = lambda x : -np.exp(g_(x))
             elif np.all(ghat>=-1e-13):
                 g_ = spline(K[ghat>0],np.log(ghat[ghat>0]),k=1)
                 ghat_spline = lambda x : np.exp(g_(x))
             else:
                 g_ = spline(K,ghat,k=1)
                 ghat_spline = g_
```

```
    if N_back != N_forward or h_back != h_forward:
        ht2 = SymmetricFourierTransform(ndim=ndim, N=N_back, h=h_back)
    else:
        ht2 = ht

    if ret_everything:
        g, g_cumsum = ht2.transform(ghat_spline, r, ret_err=False, inverse=True, ret_cumsum=
    else:
        g = ht2.transform(ghat_spline, r, ret_err=False, inverse=True)

    if ret_everything:
        return g, g_cumsum, fhat,fhat_cumsum, ghat, ht,ht2, ghat_spline
    else:
        return g
```

Forward Laplacian

We can simply use the defined functions to determine the foward laplacian in each case. We just need to specify the grid.

```
In [49]: L = 10.
         N = 256
         dr = L/N

         x_ft = np.linspace(-L+dr/2,L-dr/2,2*N)
         r_ht = np.linspace(dr/2,L-dr/2,N)
```

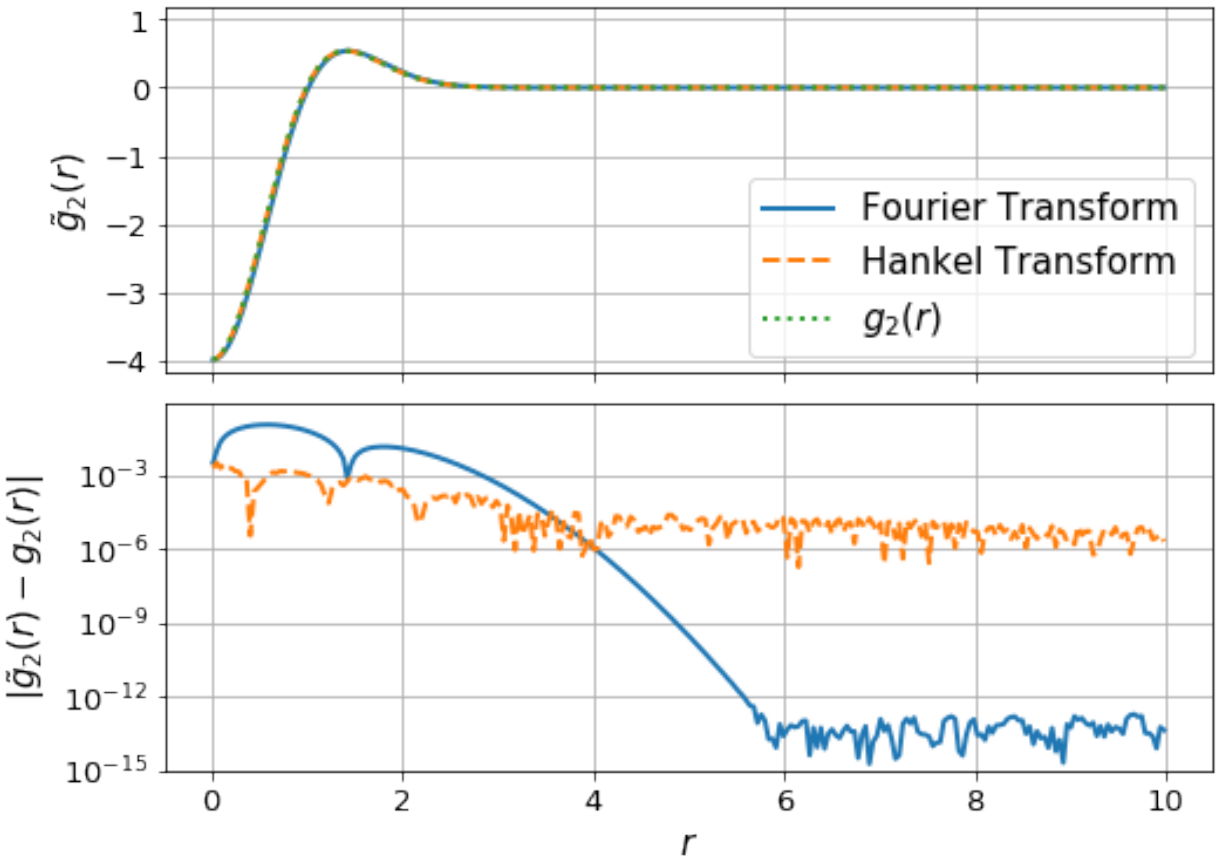
We also need to choose appropriate parameters for the forwards/backwards Hankel Transforms. To do this, we can use the `get_h` function in the `hankel` library:

```
In [50]: from hankel import get_h

         hback, res, Nback = get_h(ghat, nu=2, K=r_ht[::10], cls=SymmetricFourierTransform, atol=1e-8
         K = np.logspace(-2, 2, N) # These values come from inspection of the plot above, which shows
         hforward, res, Nforward = get_h(f, nu=2, K=K[::50], cls=SymmetricFourierTransform, atol=1e-8
         hforward, Nforward, hback, Nback

Out[50]: (9.765625e-05, 2207, 0.000390625, 534)

In [51]: ## FT
         r_ft, g_ft = ft_transformation_2d(f,x_ft)

         # Note: r_ft is equivalent to r_ht

         ## HT
         g_ht = ht_transformation_nd(f,N_forward=Nforward, h_forward=hforward, N_back=Nback, h_back=h
```

Now we plot the calculated functions against the analytic result:

```
In [52]: fig, ax = plt.subplots(2,1, sharex=True,gridspec_kw={"hspace":0.08},figsize=(8,6))

         ax[0].plot(r_ft,g_ft, label="Fourier Transform", lw=2)
         ax[0].plot(r_ht, g_ht, label="Hankel Transform", lw=2, ls='--')
         ax[0].plot(r_ht, g(r_ht), label = "$g_2(r)$", lw=2, ls = ':')
         ax[0].legend(fontsize=15)

         #ax[0].xaxis.set_ticks([])
         ax[0].grid(True)
         ax[0].set_ylabel(r"$\tilde{g}_2(r)$",fontsize=15)
```

```
ax[0].set_ylim(-4.2,1.2)

ax[1].plot(r_ft, np.abs(g_ft-g(r_ft)), lw=2)
ax[1].plot(r_ht, np.abs(g_ht-g(r_ht)),lw=2, ls='--')
#ax[1].set_ylim(-1,1)
ax[1].set_yscale('log')
#ax[1].set_yscale("symlog",linthreshy=1e-6)
ax[1].set_ylabel(r"$|\tilde{g}_2(r)-g_2(r)|$",fontsize=15)
ax[1].set_xlabel(r"$r$",fontsize=15)
ax[1].set_ylim(1e-15, 0.8)
plt.grid(True)

if prefix:
    fig.savefig(path.join(prefix,"forward_laplacian.pdf"))
```



Timing for each calculation:

```
In [53]: %timeit ft_transformation_2d(f,x_ft)
         %timeit ht_transformation_nd(f,N_forward=Nforward, h_forward=hforward, N_back=Nback, h_back=

10 loops, best of 3: 17 ms per loop
100 loops, best of 3: 19.7 ms per loop
```

Inverse Laplacian

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

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

$$2. Differentiate in Spectral space$$

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

$$3. Compute the Inverse Transform$$

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

Again, we compute the relevant Hankel parameters:

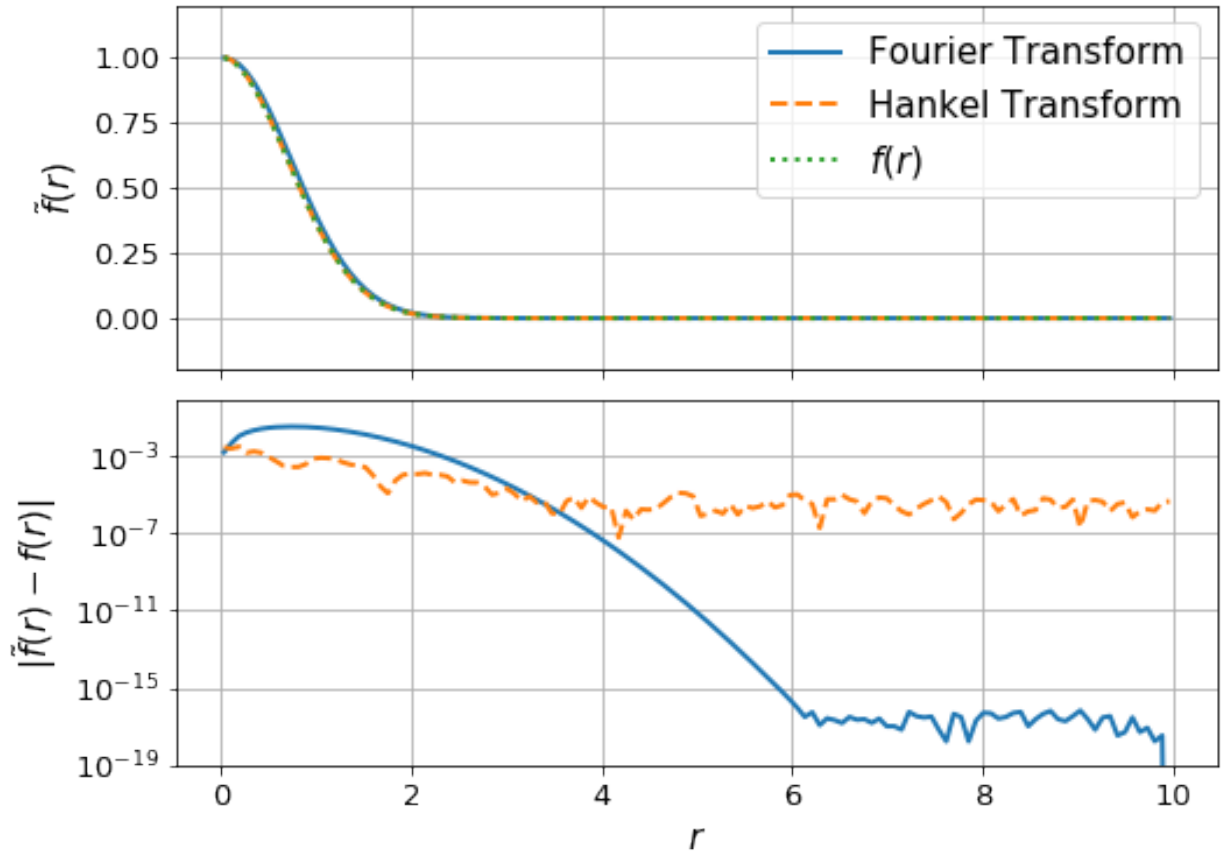
```
In [35]: hback, res, Nback = get_h(fhat, nu=2, K=r_ht[::10], cls=SymmetricFourierTransform, atol=1e-8
         K = np.logspace(-2, 2, N) # These values come from inspection of the plot above, which shows
         hforward, res, Nforward = get_h(g, nu=2, K=K[::50], cls=SymmetricFourierTransform, atol=1e-8
         hforward,Nforward,hback,Nback
```

```
Out[35]: (4.8828125e-05, 1266, 0.00078125, 375)
```

```
In [36]: ## FT
         r_ft, f_ft = ft_transformation_2d(g,x_ft, inverse=True)

         # Note: r_ft is equivalent to r_ht

         ## HT
         f_ht = ht_transformation_nd(g,N_forward=Nforward, h_forward=hforward,N_back=Nback, h_back=hb
```

```
/home/steven/miniconda3/envs/hankel2/lib/python2.7/site-packages/ipykernel/__main__.py:15: RuntimeWar
```

```
In [37]: fig, ax = plt.subplots(2,1, sharex=True,gridspec_kw={"hspace":0.08},figsize=(8,6))
         #np.mean(f(r_ft)) - np.mean(f_ft)
         ax[0].plot(r_ft,f_ft + f(r_ft)[-1] - f_ft[-1], label="Fourier Transform", lw=2)
         ax[0].plot(r_ht, f_ht, label="Hankel Transform", lw=2, ls='--')
         ax[0].plot(r_ht, f(r_ht), label = "$f(r)$", lw=2, ls = ':')
         ax[0].legend()

         ax[0].grid(True)
         ax[0].set_ylabel(r"$\tilde{f}(r)$",fontsize=15)
         ax[0].set_ylim(-0.2,1.2)
         #ax[0].set_yscale('log')

         ax[1].plot(r_ft, np.abs(f_ft + f(r_ft)[-1] - f_ft[-1] -f(r_ft)), lw=2)
         ax[1].plot(r_ht, np.abs(f_ht -f(r_ht)),lw=2, ls='--')

         ax[1].set_yscale('log')
         ax[1].set_ylabel(r"$|\tilde{f}(r)-f(r)|$",fontsize=15)
         ax[1].set_xlabel(r"$r$",fontsize=15)
         ax[1].set_ylim(1e-19, 0.8)
         plt.grid(True)

         if prefix:
             fig.savefig(path.join(prefix,"inverse_laplacian.pdf"))
```




```
In [28]: %timeit ft_transformation_2d(g,x_ft, inverse=True)
         %timeit ht_transformation_nd(g,N_forward=Nforward, h_forward=hforward,N_back=Nback, h_back=h
```

```
/home/steven/miniconda3/envs/hankel2/lib/python2.7/site-packages/ipykernel/__main__.py:15: RuntimeWa
```

```
10 loops, best of 3: 19.3 ms per loop
10 loops, best of 3: 30.6 ms per loop
```

3D Problem (Forward)

We need to define the FT function again, for 3D:

```
In [38]: def ft_transformation_3d(f,x, inverse=False):

             r = np.sqrt(np.sum(np.array(np.meshgrid(*([x]*3)))**2,axis=0))

             # Appropriate k-space values
             k = 2*np.pi*fftfreq(len(x),d=x[1]-x[0])
             K2 = np.sum(np.array(np.meshgrid(*([k]*3)))**2,axis=0)

             # The transformation
             if not inverse:
                 g2d = ifftn(-K2 * fftn(f(r)).real).real
             else:
                 invK2 = 1./K2
                 invK2[np.isinf(invK2)] = 0.0

                 g2d = ifftn(-invK2 * fftn(f(r)).real).real
```

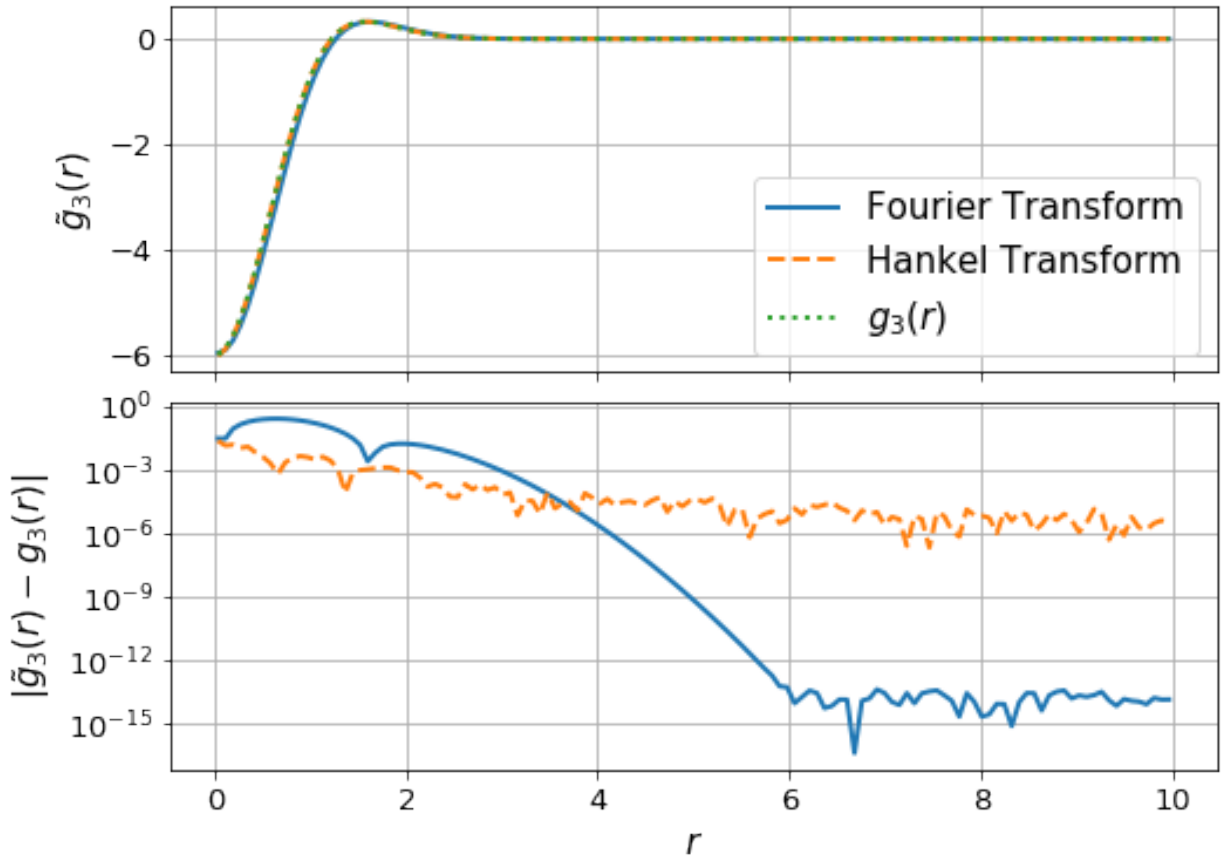
```
            return x[len(x)/2:], g2d[len(x)/2,len(x)/2, len(x)/2:]
```

We also need to define the 3D laplacian function:

```
In [39]: g3 = lambda r: 4.0*np.exp(-r**2)*(r**2 - 1.5)
         fhat_3d = lambda x : np.pi**(3./2)*np.exp(-x**2/4.)
         ghat_3d = lambda x : -x**2*fhat_3d(x)

In [40]: L = 10.
         N = 128
         dr = L/N

         x_ft = np.linspace(-L+dr/2,L-dr/2,2*N)
         r_ht = np.linspace(dr/2,L-dr/2,N)
```

Again, choose our resolution parameters

```
In [32]: hback, res, Nback = get_h(ghat_3d, nu=3, K=r_ht[::10], cls=SymmetricFourierTransform, atol=1
         K = np.logspace(-2, 2, 2*N) # These values come from inspection of the plot above, which sho
         hforward, res, Nforward = get_h(f, nu=3, K=K[::50], cls=SymmetricFourierTransform, atol=1e-8
         hforward, Nforward, hback, Nback

Out[32]: (9.765625e-05, 2207, 0.00078125, 375)

In [33]: ## FT
         r_ft, g_ft = ft_transformation_3d(f,x_ft)

         # Note: r_ft is equivalent to r_ht

         ## HT
         K = np.logspace(-1.0,2.,N)
         g_ht = ht_transformation_nd(f,N_forward=Nforward, h_forward=hforward, N_back=Nback, h_back=h

In [65]: fig, ax = plt.subplots(2,1, sharex=True,gridspec_kw={"hspace":0.08},figsize=(8,6))

         ax[0].plot(r_ft,g_ft, label="Fourier Transform", lw=2)
         ax[0].plot(r_ht, g_ht, label="Hankel Transform", lw=2, ls='--')
         ax[0].plot(r_ht, g3(r_ht), label = "$g_3(r)$", lw=2, ls = ':')
         ax[0].legend(fontsize=15)

         #ax[0].xaxis.set_ticks([])
         ax[0].grid(True)
         ax[0].set_ylabel(r"$\tilde{g}_3(r)$",fontsize=15)
         #ax[0].set_ylim(-4.2,1.2)

         ax[1].plot(r_ft, np.abs(g_ft-g3(r_ft)), lw=2)
         ax[1].plot(r_ht, np.abs(g_ht-g3(r_ht)),lw=2, ls='--')

         ax[1].set_yscale('log')
         ax[1].set_ylabel(r"$|\tilde{g}_3(r)-g_3(r)|$",fontsize=15)
         ax[1].set_xlabel(r"$r$",fontsize=15)
         plt.grid(True)

         if prefix:
             fig.savefig(path.join(prefix,"forward_laplacian_3D.pdf"))
```



In [72]:
```
%timeit ht_transformation_nd(f,N_forward=Nforward, h_forward=hforward, N_back=Nback, h_back=

%timeit ft_transformation_3d(f,x_ft)
```

```
100 loops, best of 3: 16.3 ms per loop
1 loop, best of 3: 2.66 s per loop
```

3D Problem (Inverse)

In [41]:
```
hback, res, Nback = get_h(fhat_3d, nu=3, K=r_ht[::10], cls=SymmetricFourierTransform, atol=1
K = np.logspace(-2, 2, N) # These values come from inspection of the plot above, which shows
hforward, res, Nforward = get_h(g3, nu=3, K=K[::50], cls=SymmetricFourierTransform, atol=1e-
hforward,Nforward,hback,Nback
```

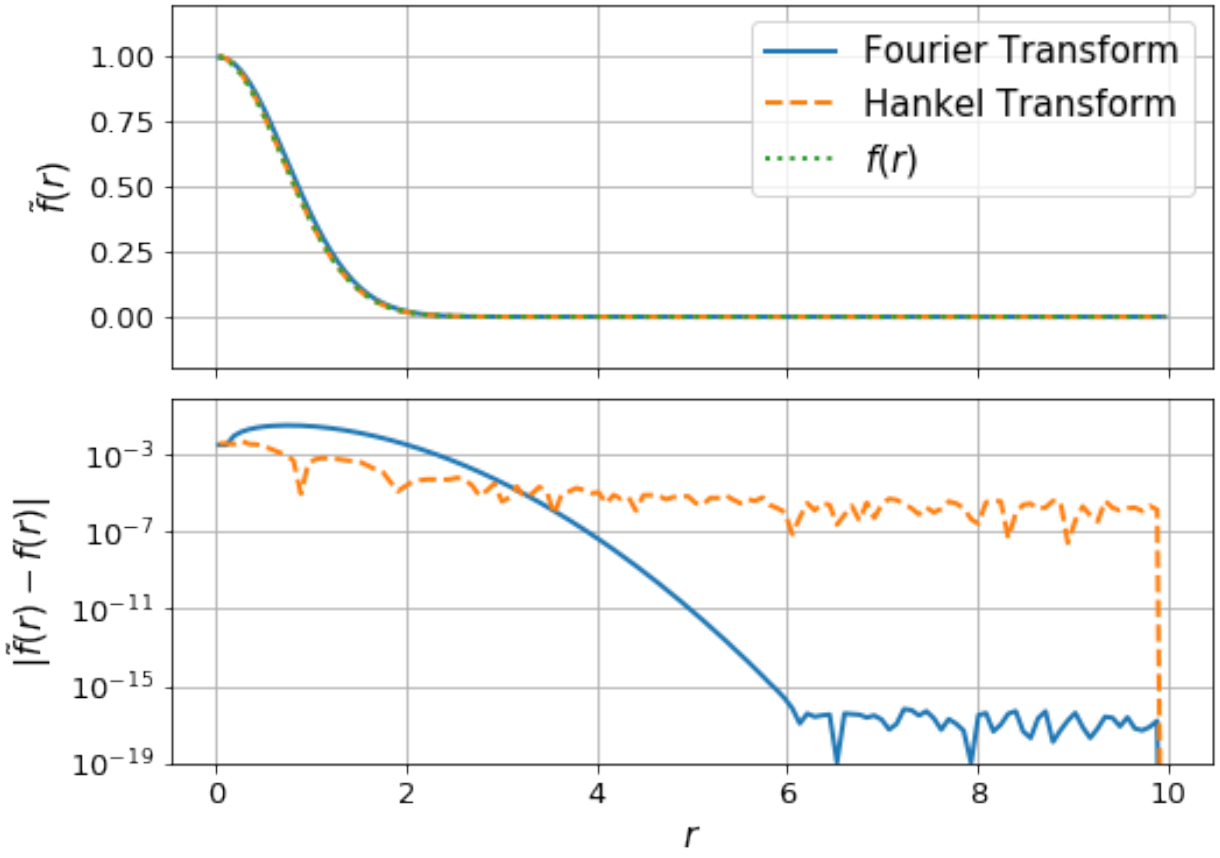
Out[41]: (6.103515625e-06, 3576, 0.00078125, 375)

In [46]:
```
## FT
r_ft, f_ft = ft_transformation_3d(g3,x_ft, inverse=True)

# Note: r_ft is equivalent to r_ht

## HT
f_ht = ht_transformation_nd(g3,ndim=3, N_forward=Nforward, h_forward=hforward,N_back=Nback,
```

```
/home/steven/miniconda3/envs/hankel2/lib/python2.7/site-packages/ipykernel/__main__.py:13: RuntimeWa
```

In [48]:
```
fig, ax = plt.subplots(2,1, sharex=True,gridspec_kw={"hspace":0.08},figsize=(8,6))
#np.mean(f(r_ft)) - np.mean(f_ft)
ax[0].plot(r_ft, f_ft + f(r_ft)[-1] - f_ft[-1], label="Fourier Transform", lw=2)
```

```
ax[0].plot(r_ht, f_ht + f(r_ft)[-1] - f_ht[-1], label="Hankel Transform", lw=2, ls='--')
ax[0].plot(r_ht, f(r_ht), label = "$f(r)$", lw=2, ls = ':')
ax[0].legend()

ax[0].grid(True)
ax[0].set_ylabel(r"$\tilde{f}(r)$",fontsize=15)
ax[0].set_ylim(-0.2,1.2)
#ax[0].set_yscale('log')

ax[1].plot(r_ft, np.abs(f_ft + f(r_ft)[-1] - f_ft[-1] -f(r_ft)), lw=2)
ax[1].plot(r_ht, np.abs(f_ht + f(r_ft)[-1] - f_ht[-1] -f(r_ht)),lw=2, ls='--')

ax[1].set_yscale('log')
ax[1].set_ylabel(r"$|\tilde{f}(r)-f(r)|$",fontsize=15)
ax[1].set_xlabel(r"$r$",fontsize=15)
ax[1].set_ylim(1e-19, 0.8)
plt.grid(True)

if prefix:
    fig.savefig(path.join(prefix,"inverse_laplacian_3d.pdf"))
```



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

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

and we recall that $y_{\nu k}$, ψ, ψ' and $w_{\nu k}$ are

$$y_{\nu k} = \pi\psi(hr_{\nu k})/h \tag{5.3}$$
$$\psi(t) = t\tanh(\pi\sinh(t)/2) \tag{5.4}$$
$$\psi'(t) = \frac{\pi t\cosh(t) + \sinh(\pi\sinh(t))}{1+\cosh(\pi\sinh(t))} \tag{5.5}$$
$$w_{\nu k} = \frac{Y_\nu(\pi r_{\nu k})}{J_{\nu+1}(\pi r_{\nu k})}. \tag{5.6}$$

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 $hr_{\nu k}$.

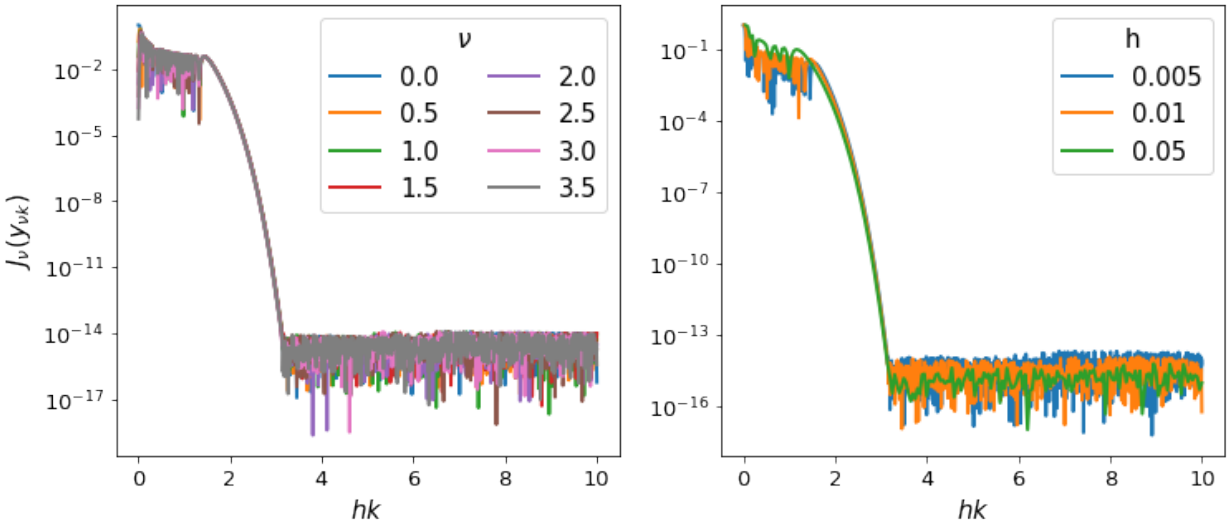
```
In [1]: from scipy.special import yv,jv
        from mpmath import fp as mpm
        import numpy as np

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

        from hankel import HankelTransform, SymmetricFourierTransform
```

```
In [2]: mpl.rcParams['lines.linewidth'] = 2
        mpl.rcParams['xtick.labelsize'] = 13
        mpl.rcParams['ytick.labelsize'] = 13
        mpl.rcParams['font.size'] = 15
        mpl.rcParams['axes.titlesize'] = 14
```

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

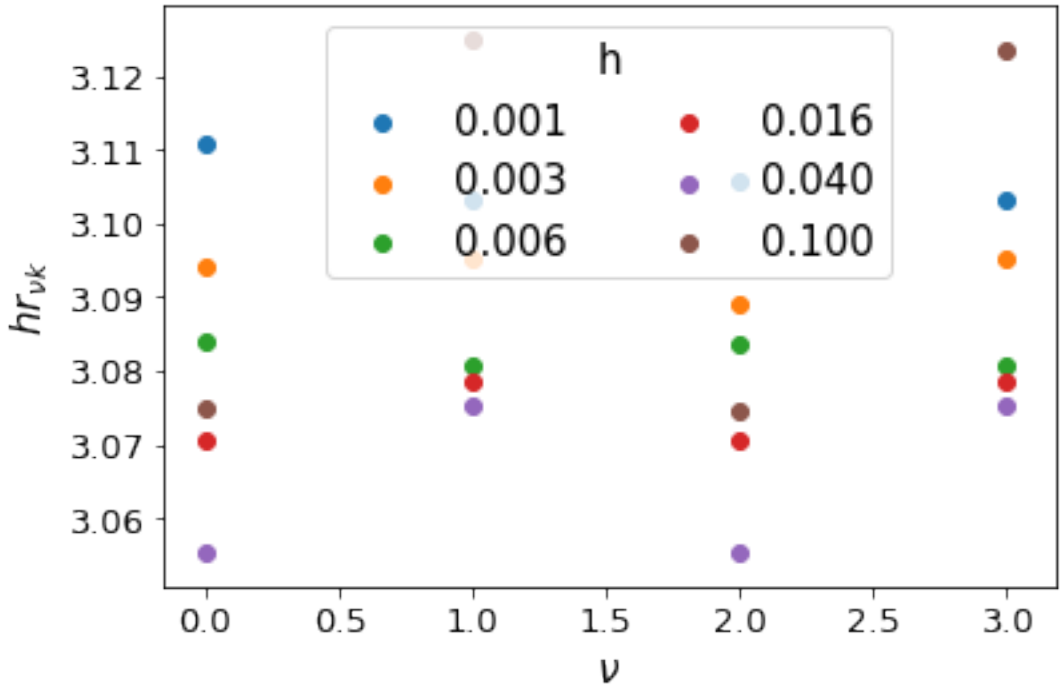
```
In [15]: fig,ax = plt.subplots(1,2,figsize=(12,5), subplot_kw={"yscale":'log'})
         for nu in np.arange(0,4,0.5):
             ht= HankelTransform(nu=nu,N=1000, h = 0.01)
             ax[0].plot(ht._h*np.arange(1,1001), np.abs(jv(ht._nu, ht.x)), label=str(nu))

         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"$hk$")


    plt.savefig("/home/steven/Documents/Projects/HANKEL/laplacian_paper/Figures/h_N_convergence
```



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:

```
In [4]: for i,nu in enumerate( np.arange(0,4)):
            for j,h in enumerate(np.logspace(-3,-1,6)):
                ht= HankelTransform(nu=nu,N=int(5./h), h = h)
                plt.scatter(nu,ht._h*ht._zeros[np.where(np.abs(jv(ht._nu, ht.x))<1e-13)[0][0]],color=

        plt.xlabel(r"$\nu$")
        plt.ylabel(r"$hr_{\nu k}$")
        plt.legend(title="h",ncol=2)

Out[4]: <matplotlib.legend.Legend at 0x7f1e3ac52550>
```



Clearly, we can cut the summation at $hr_{\nu k} = 3.2$ 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. $N = 3.2/h$.

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. 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.

The effect that h has, given its relation to N above, is to stretch the domain of the integration. At large k, the nodes are given by $y_{\nu k} \approx \pi k$, so that the maximum point evaluated on $f(x)$ is at $x = 3.2\pi/h$. Given some knowledge of f, we can therefore set some limits on the value of h.

For instance, the N^{th} term of the sum (if N is high enough) is approximately

$$G_\nu \approx \pi f(3.2\pi/h) J_\nu(3.2\pi/h). \tag{5.8}$$

For large arguments, J_ν approaches a cosine function:

$$J_\nu(x) \approx \sqrt{\frac{2}{\pi x}} \left(\cos\left(x - \frac{\nu\pi}{2} - \frac{\pi}{4}\right)\right). \tag{5.9}$$

Taking the amplitude of this function yields

$$G_\nu \approx \sqrt{\frac{2h}{3.2}} f(3.2\pi/h). \tag{5.10}$$

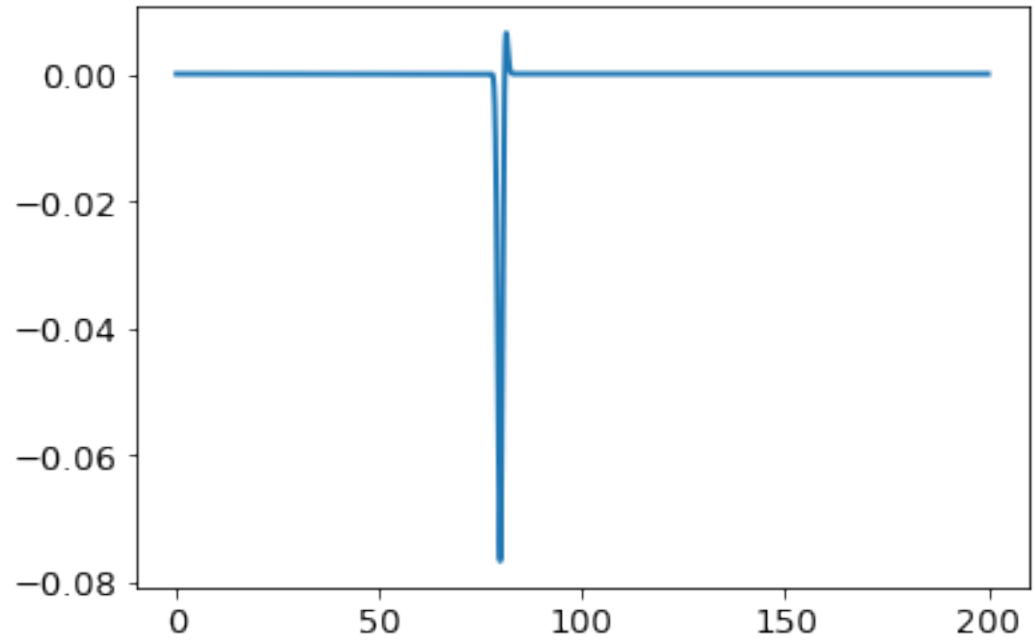
One obvious criterion for h is that dG/dh must be negative, so that the sum is probing the convergent part of the integral. However, the most salient criterion for h is that it successfully samples the smallest features in $f(x)$. Thus for instance, if $f(x)$ is a narrow Gaussian centred at some x_0, then h must be chosen so as to cover the FWHM of the Gaussian adequately. Alternatively, if $f(x)$ has a lot of information close to $x = 0$, then h must be chosen so as to ensure several nodes cover that information.

Unfortunately, these criteria are not easily solvable, and so we suggest iteratively modifying h until convergence is reached.

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

```
In [3]: x = np.linspace(0,200.,1000000)
        ff = lambda x : np.exp(-(x-80.)**2/1.)
        plt.plot(x,ff(x) * jv(0,x))

        from scipy.integrate import simps
        print "Integral is: ", simps(ff(x) * jv(0,x),x)
```

```
Integral is:  -0.0965117065719
```

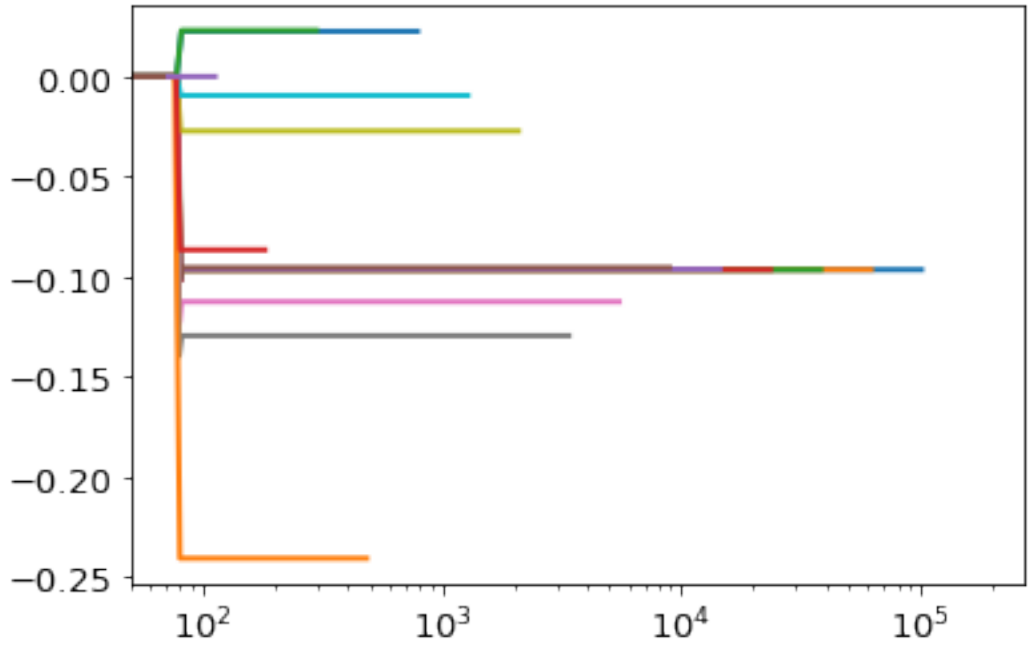



```
In [6]: for h in np.logspace(-4,0,20):
            N = int(3.2/h)
            ht = HankelTransform(nu=0, h=h, N=N)
            G = ht.G(ff,h)
            ans,cum  = ht.integrate(f= ff,ret_cumsum=True,ret_err=False)
            print np.pi*3.2/h, N>25, G, ans, np.sum(np.logical_and(ht.x>78,ht.x<82))

            plt.plot(ht.x,cum)
        plt.xscale('log')
        plt.xlim(50,)
```

```
100530.964915 True 0.0 -0.0965117065719 10
61911.8148996 True 0.0 -0.0965117065719 8
38128.2804497 True 0.0 -0.0965117058915 6
23481.233306 True 0.0 -0.0965121903118 5
14460.8755251 True 0.0 -0.0962742533513 4
8905.7043226 True 0.0 -0.0955164471316 3
```

```
5484.56207535 True 0.0 -0.112512671321 2
3377.65774259 True 0.0 -0.129518816033 2
2080.12447837 True 0.0 -0.027210019158 2
1281.0409388 True 0.0 -0.00944251806079 1
788.92677046 True 0.0 0.0226151150847 1
485.85914025 True 0.0 -0.240786142962 1
299.215482353 True 0.0 0.0230365521459 1
184.27131953 True 0.0 -0.0866049560486 1
113.483162484 True 0.0 -4.70925410083e-06 1
69.8884026016 False 1.18224587086e-45 -5.00649306791e-74 0
43.0406477165 False 0.0 0.0 0
26.5065058993 False 0.0 0.0 0
16.3239842397 False 0.0 0.0 0
10.0530964915 False 0.0 0.0 0
```

Out[6]: (50, 268753.68203171075)

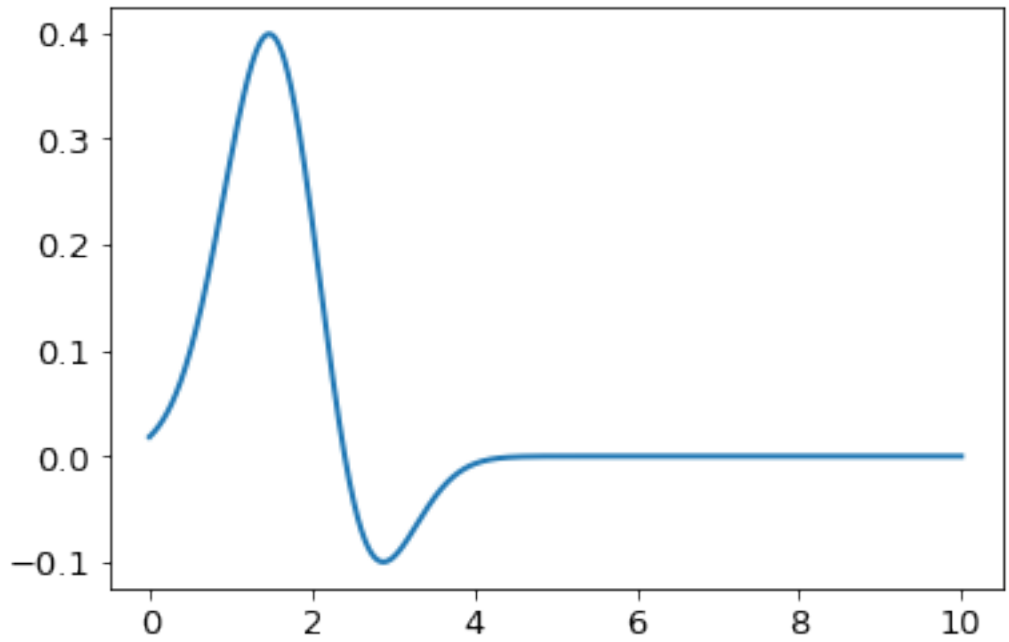


In the above example we see that only for very large values of h was the criteria of negative derivative not met. However, the criteria of covering the interesting region with nodes was only met by the smallest 5 values of h, each of which yields a good value of the integral.

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

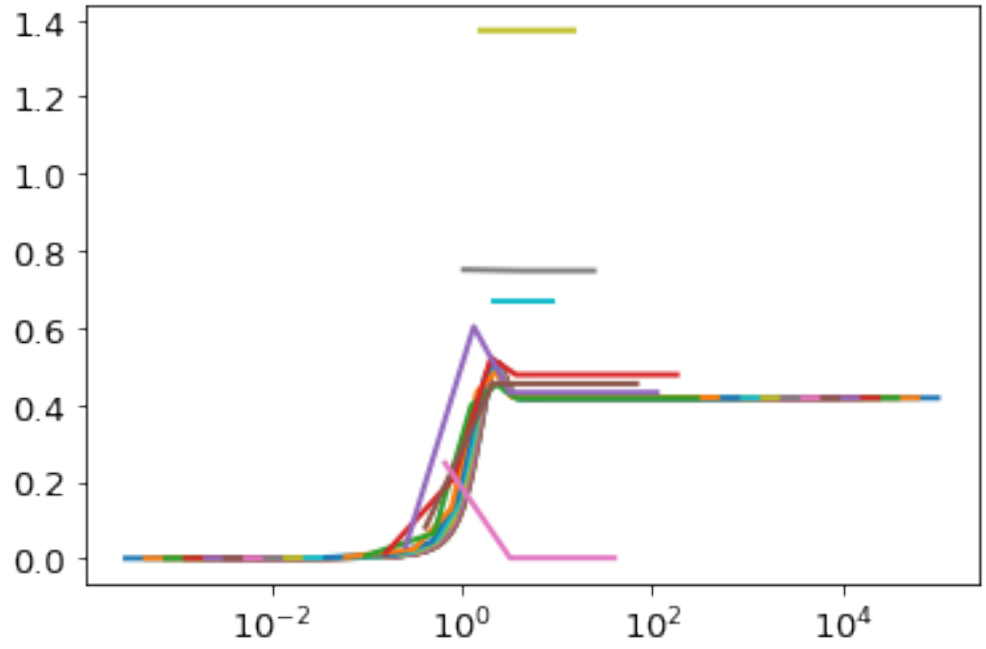
```
In [12]: x = np.linspace(0,10,10000)
         ff = lambda x : np.exp(-(x-2)**2/1.)
         plt.plot(x,ff(x) * jv(0,x))

         from scipy.integrate import simps
         print "Integral is: ", simps(ff(x) * jv(0,x),x)
```

```
Integral is:  0.416843377992
```




```
In [8]: for h in np.logspace(-4,0,20):
            N = int(3.2/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 np.pi*3.2/h, N>25, G, ans, np.sum(np.logical_and(ht.x>78,ht.x<82))

            plt.plot(ht.x,cum)
        plt.xscale('log')
```

```
100530.964915 True 0.0 0.416843377981 10
61911.8148996 True 0.0 0.416843377981 8
38128.2804497 True 0.0 0.416843377981 6
23481.233306 True 0.0 0.416843377981 5
14460.8755251 True 0.0 0.416843377981 4
8905.7043226 True 0.0 0.416843377981 3
5484.56207535 True 0.0 0.416843377981 2
3377.65774259 True 0.0 0.416843377981 2
2080.12447837 True 0.0 0.416843377721 2
1281.0409388 True 0.0 0.416842995677 1
788.92677046 True 0.0 0.416785363191 1
485.85914025 True 0.0 0.418317506223 1
299.215482353 True 0.0 0.416384483699 1
184.27131953 True 0.0 0.477328299594 1
113.483162484 True 0.0 0.431203499939 1
69.8884026016 False 0.0 0.453832154083 0
43.0406477165 False 0.0 0.00104616438595 0
26.5065058993 False 7.30607651209e-262 0.747476886413 0
16.3239842397 False 4.84891358879e-90 1.37191780468 0
10.0530964915 False 5.40650458253e-29 0.670764719471 0
```



Here we are able to achieve good precision with just ~800 terms.

These ideas are built into the `get_h` function in `hankel`:

```
In [16]: from hankel import get_h

In [17]: get_h(lambda x : np.exp(-(x-2)**2/1.),0)

Out[17]: (0.00625, 0.41684338301151935, 31)

In [55]: get_h(lambda x : np.exp(-(x-80.)**2),0)

Out[55]: (0.000390625, -0.096511400178473233, 236)

In [56]: get_h(lambda x : np.exp(-x**2),0,atol=1e-10,rtol=1e-8)

Out[56]: (0.0125, 0.78515069875817434, 21)
```

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 know 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.

```
In [57]: get_h(lambda x : np.exp(-(x-80.)**2),2, K= np.logspace(-2,2,10), cls=SymmetricFourierTransfo

Out[57]: (6.103515625e-06, array([  7.53952457e+02,   8.54080244e+01,   1.78300196e+02,
                -1.25465226e+02,  -9.37124420e+01,   2.30734389e+01,
                 1.67032541e-01,  -2.24867894e-13,   2.34702637e-14,
                 1.25747785e-14]), 18957)

In [58]: get_h(lambda x : np.exp(-(x-80.)**2),2, K= np.logspace(-2,2,10), cls=SymmetricFourierTransfo
```

```
Out[58]: (5e-06, array([  7.53952144e+02,   8.54080244e+01,   1.78300196e+02,
                 -1.25465226e+02,  -9.37124420e+01,   2.30734389e+01,
                  1.67032541e-01,   8.68147276e-13,  -1.15869805e-13,
                 -5.97366010e-13]), 20928)
```

For a more sane example

```
In [61]: get_h(lambda x : np.exp(-x**2), 2, K= np.logspace(-2,2,10), cls=SymmetricFourierTransform)
```

```
Out[61]: (9.765625e-05, array([  3.14151417e+00,   3.14098461e+00,   3.13688784e+00,
                   3.10534840e+00,   2.87164359e+00,   1.56688573e+00,
                   1.43880563e-02,   1.70297152e-16,  -8.17366143e-17,
                  -1.00538714e-16]), 2423)
```

Simple Exponential in 2D

Specifically for the transform of e^{-r^2}, we know the intermediate form: $-\pi k^2 e^{-k^2/4}$.

```
In [52]: f = lambda x : np.exp(-x**2)
         fhat = lambda x : np.pi*np.exp(-x**2/4.)
         g = lambda x : 4*np.exp(-x**2) * (x**2 - 1)
         ghat = lambda x : -x**2*fhat(x)
```

Thus for our problem in particular, we can more easily choose the inputs. First, let's define the final set of r_i:

```
In [53]: rfinal = np.logspace(-2,1,50)
```

We begin by generating the backwards parameters, which should give us an indication of what range we need to evaluate $\hat{g}(K)$ on. We down-sample the input range of scales because we just care about getting the correct parameters across the board.

```
In [62]: hback, res, Nback = get_h(lambda x : -np.pi*x**2 * np.exp(-x**2/4.), nu=2, K=rfinal[::5], cl
         hback, Nback, res
```

```
Out[62]: (0.0001953125,
          570,
          array([ -3.99919897e+00,  -3.99672507e+00,  -3.98660219e+00,
                  -3.94535012e+00,  -3.77974337e+00,  -3.15449842e+00,
                  -1.31932539e+00,   5.39979504e-01,   1.01797788e-02,
                   1.09299479e-12]))
```

We can compare this to the truth:

```
In [63]: g(rfinal[::5])
```

```
Out[63]: array([ -3.99920006e+00,  -3.99672507e+00,  -3.98660219e+00,
                 -3.94535012e+00,  -3.77974337e+00,  -3.15449842e+00,
                 -1.31932539e+00,   5.39979504e-01,   1.01797788e-02,
                  1.09279421e-12])
```

Given that we know we have to use the K given by the h and N we derived here, we can evaluate these:

```
In [64]: sft = SymmetricFourierTransform(2, h=hback, N=Nback)
         K_range = [sft.x.min()/rfinal.max(), sft.x.max()/rfinal.min()]
         K_range
```

```
Out[64]: [5.6476424591524198e-05, 31035.198303127032]
```

```
In [65]: K = np.logspace(np.log10(K_range[0]), np.log10(K_range[1]), 1000)
```

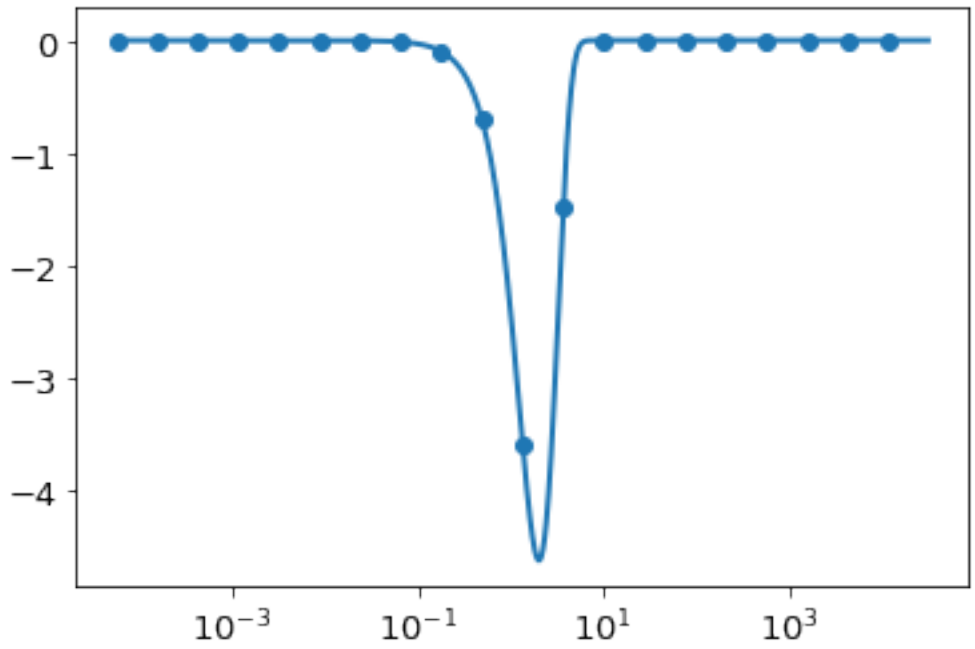
Let's try to evaluate the forward-transform, on this range of K:

```
In [66]: hforward, res, Nforward = get_h(lambda x : np.exp(-x**2), nu=2, K=K[::50], cls=SymmetricFou
         hforward, Nforward, res
```

```
Out[66]: (9.765625e-07,
          261372,
          array([  3.14157789e+00,   3.14159263e+00,   3.14159251e+00,
                   3.14159160e+00,   3.14158474e+00,   3.14153334e+00,
                   3.14114799e+00,   3.13826062e+00,   3.11669911e+00,
                   2.95970616e+00,   2.00897355e+00,   1.10017458e-01,
                   3.83886676e-11,  -1.23702908e-16,  -8.99054184e-17,
                  -7.70683509e-17,   1.02955886e-16,   3.66853159e-17,
                  -3.22748661e-17,  -4.72472102e-17]))
```

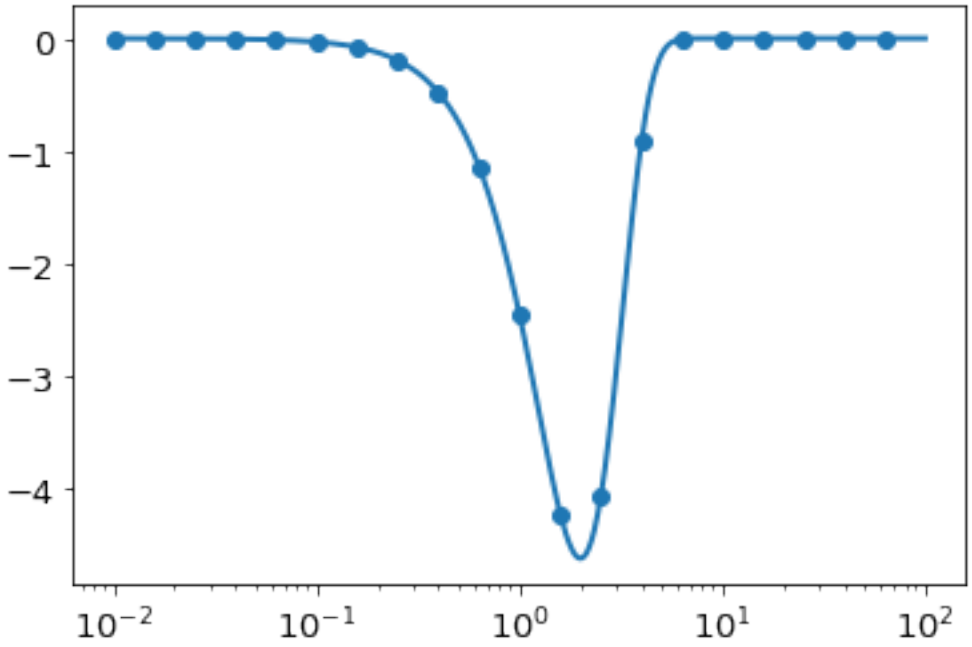
This takes a long time, and returns a very small h and high N. This is most likely because we are using very small K. Taking a look at the plot of $\hat{g}(K)$, we find that we should be able to get away with focusing on the inner part, and the spline should be very accurate towards the edges:

```
In [68]: plt.plot(K,ghat(K))
         plt.scatter(K[::50], -K[::50]**2 * res)
         plt.xscale('log')
```



Let's set the range of K smaller:

```
In [71]: K = np.logspace(-2, 2, 1000)

In [72]: hforward, res, Nforward = get_h(lambda x : np.exp(-x**2), nu=2, K=K[::50], cls=SymmetricFou
         hforward, Nforward, res

Out[72]: (0.000125, 1703, array([  3.14151411e+00,   3.14139520e+00,   3.14109623e+00,
                   3.14034468e+00,   3.13845594e+00,   3.13371228e+00,
                   3.12181741e+00,   3.09211036e+00,   3.01866366e+00,
                   2.84161861e+00,   2.44101598e+00,   1.66588629e+00,
                   6.37493106e-01,   5.69672988e-02,   1.31402547e-04,
                   3.08028340e-11,  -3.61323428e-16,  -4.02706283e-16,
                  -3.50161291e-16,  -3.27828863e-16]))

In [73]: plt.plot(K,ghat(K))
         plt.scatter(K[::50], -K[::50]**2 * res)
         plt.xscale('log')
```



Let's now try the entire forwards-then-back solution using the parameters we've optimized. First, we define a function for doing the whole transformation, and also a function for plotting a diagnostic plot:

```
In [75]: def ht_transformation_nd(f,N_forward,h_forward,K,r,ndim=2, inverse=False, N_back=None, h_bac
                                  ret_everything=False):

             if N_back is None:
                 N_back = N_forward
             if h_back is None:
                 h_back = h_forward

             # Get transform of f
             ht = SymmetricFourierTransform(ndim=ndim, N=N_forward, h=h_forward)

             if ret_everything:
                 fhat, fhat_cumsum = ht.transform(f, K, ret_cumsum=True, ret_err=False)
             else:
                 fhat = ht.transform(f, K, ret_err = False)

             # Spectral derivative
             if not inverse:
                 ghat = -K**2 * fhat
             else:
                 ghat = -1./K**2 * fhat

             # Transform back to physical space via spline
             # The following should give best resulting splines for most kinds of functions
             # Use log-space y if ghat is either all negative or all positive, otherwise linear-space
             # Use order 1 because if we have to extrapolate, this is more stable.
             # This will not be a good approximation for discontinuous functions... but they shouldn
             if np.all(ghat<=1e-13):
                 g_ = spline(K[ghat<0],np.log(-ghat[ghat<0]),k=1)
                 ghat_spline = lambda x : -np.exp(g_(x))
             elif np.all(ghat>=-1e-13):
                 g_ = spline(K[ghat<0],np.log(ghat[ghat<0]),k=1)
```

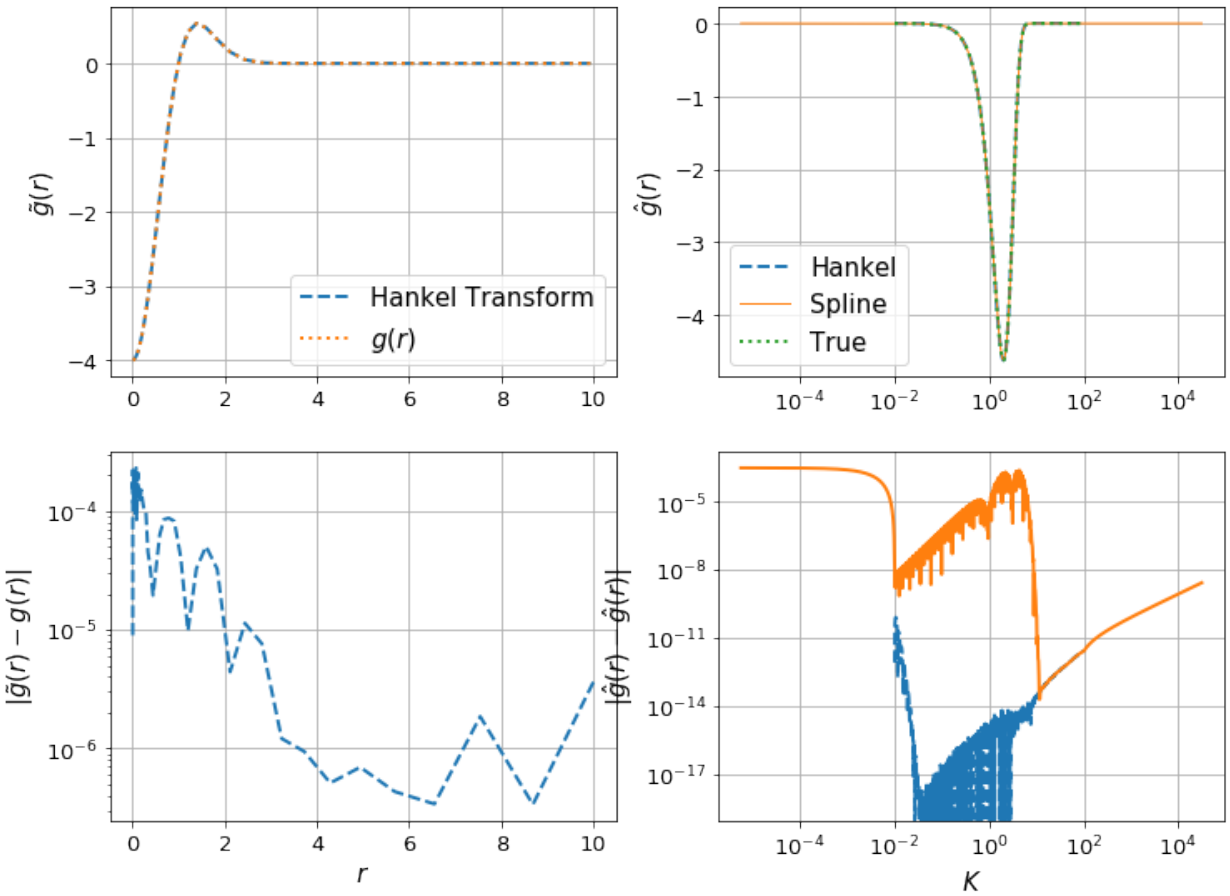
```
            ghat_spline = lambda x : np.exp(g_(x))
        else:
            g_ = spline(K,ghat,k=1)
            ghat_spline = g_

        if N_back != N_forward or h_back != h_forward:
            ht2 = SymmetricFourierTransform(ndim=ndim, N=N_back, h=h_back)
        else:
            ht2 = ht

        if ret_everything:
            g, g_cumsum = ht2.transform(ghat_spline, r, ret_err=False, inverse=True, ret_cumsum=
        else:
            g = ht2.transform(ghat_spline, r, ret_err=False, inverse=True)

        if ret_everything:
            return g, g_cumsum, fhat,fhat_cumsum, ghat, ht,ht2, ghat_spline
        else:
            return g
```

In [76]:
```
def make_diagnostic_plots(K, r, ghat=None,g=None,*args,**kwargs):
    """
    Calls ht_transformation_nd and takes "true" functions to assess the discrepancy.
    """

    g_ht, g_cumsum, fhat_ht, fhat_cumsum,ghat_ht, ht1,ht2, ghat_spline = ht_transformation_

    fig, ax = plt.subplots(2,2,figsize=(12,9))

    ## Plot g
    ax[0,0].plot(r, g_ht, label="Hankel Transform", lw=2, ls='--')
    ax[0,0].plot(r, g(r), label = "$g(r)$", lw=2, ls = ':')
    ax[0,0].legend(fontsize=15)

    ax[0,0].grid(True)
    ax[0,0].set_ylabel(r"$\tilde{g}(r)$",fontsize=15)

    ax[1,0].plot(r, np.abs(g_ht-g(r)),lw=2, ls='--')
    ax[1,0].set_yscale('log')
    ax[1,0].set_ylabel(r"$|\tilde{g}(r)-g(r)|$",fontsize=15)
    ax[1,0].set_xlabel(r"$r$",fontsize=15)
    ax[1,0].grid(True)


    ## Plot ghat
    ax[0,1].plot(K, ghat_ht, label="Hankel", lw=2, ls='--')

    allk = np.logspace(np.log10(ht2.xrange(K)[0]), np.log10(ht2.xrange(K)[1]),1000)
    ax[0,1].plot(allk, ghat_spline(allk), label="Spline",lw=1)
    ax[0,1].plot(K, ghat(K), label = "True", lw=2, ls = ':')
    ax[0,1].legend(fontsize=15)

    ax[0,1].grid(True)
    ax[0,1].set_ylabel(r"$\hat{g}(r)$",fontsize=15)
    ax[0,1].set_xscale('log')

    ax[1,1].plot(K, np.abs(ghat_ht-ghat(K)),lw=2, ls='--')
    ax[1,1].plot(allk, np.abs(ghat_spline(allk)-ghat(allk)),lw=2, ls='-')
```

```
        ax[1,1].set_yscale('log')
        ax[1,1].set_ylabel(r"$|\hat{g}(r)-\hat{g}(r)|$",fontsize=15)
        ax[1,1].set_xlabel(r"$K$",fontsize=15)
        ax[1,1].grid(True)
        ax[1,1].set_xscale('log')
        return fig
```

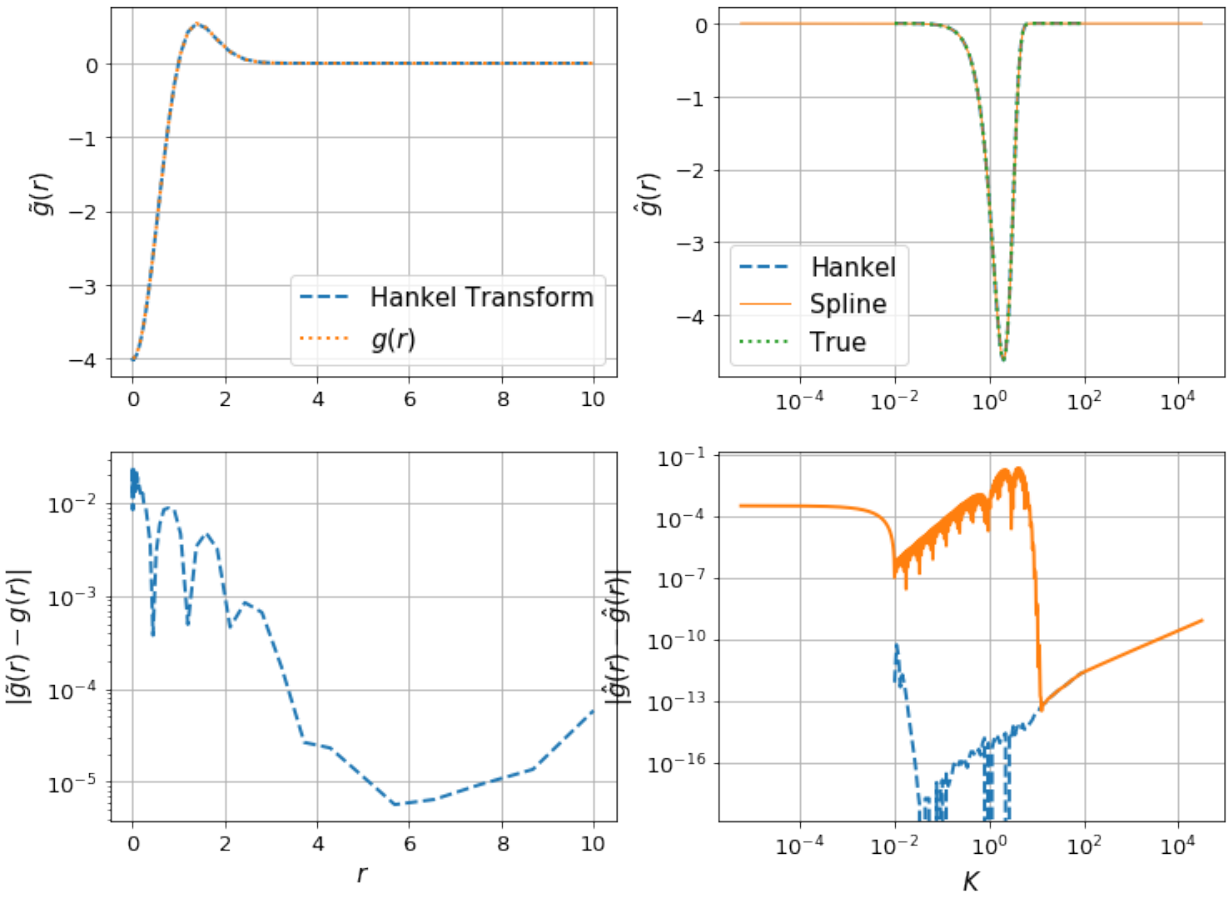
```
In [92]: g_ht, g_cumsum, fhat_ht, fhat_cumsum,ghat_ht, ht1,ht2, ghat_spline = ht_transformation_nd(la
```

```
In [80]: make_diagnostic_plots(K, rfinal, ghat, g, f=lambda x : np.exp(-x**2),N_forward=Nforward, h_f
```



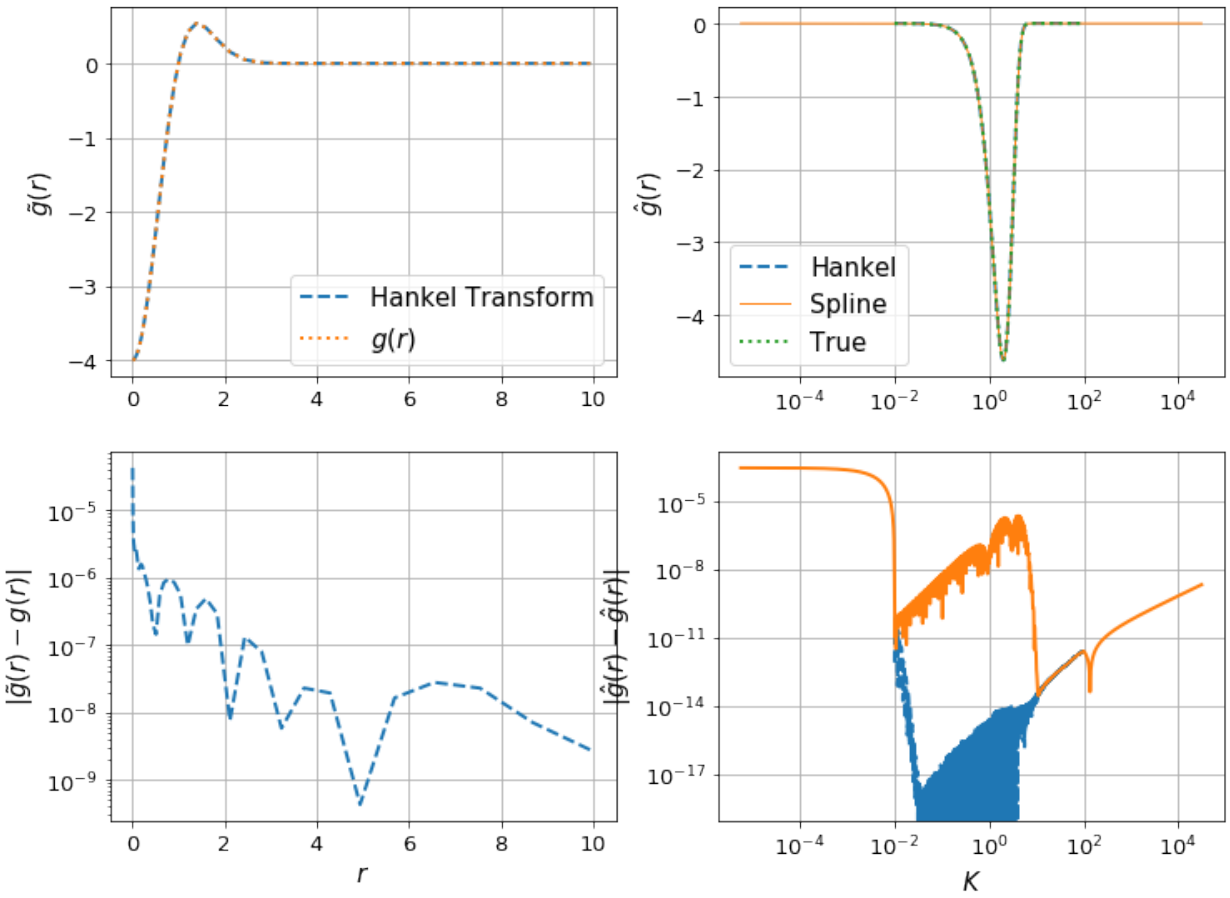
Let's check the effect of the resolution of K:

```
In [82]: K = np.logspace(-2,2,100)
         make_diagnostic_plots(K, rfinal, ghat, g, f=lambda x : np.exp(-x**2),N_forward=Nforward, h_f
```



The difference can be clearly seen in the orange line in the bottom right plot – the spline evaluation of $\hat{g}$ is less accurate, and so we have a corresponding drop in accuracy in $g(r)$.

```
In [83]: K = np.logspace(-2,2,10000)
         make_diagnostic_plots(K, rfinal, ghat, g, f=lambda x : np.exp(-x**2),N_forward=Nforward, h_
```



Here we have an increase in precision, by a factor of 10, with an increase in n_k by a factor of 10. We'd probably gain a bit by using a higher-order spline interpolation, however this has bad behaviour when extrapolating, which we need to do here (otherwise we need to use extremely large N to get the highest K).

SImple Exponential in 3D

In this section, we perform the same analysis as in the previous one, but in 3D, to show that the general process works well.

```
In [84]: fhat_3d = lambda x : np.pi**(3./2)*np.exp(-x**2/4.)
         g_3d = lambda r: 4.0*np.exp(-r**2)*(r**2 - 1.5)
         ghat_3d = lambda x : -x**2*fhat_3d(x)
```

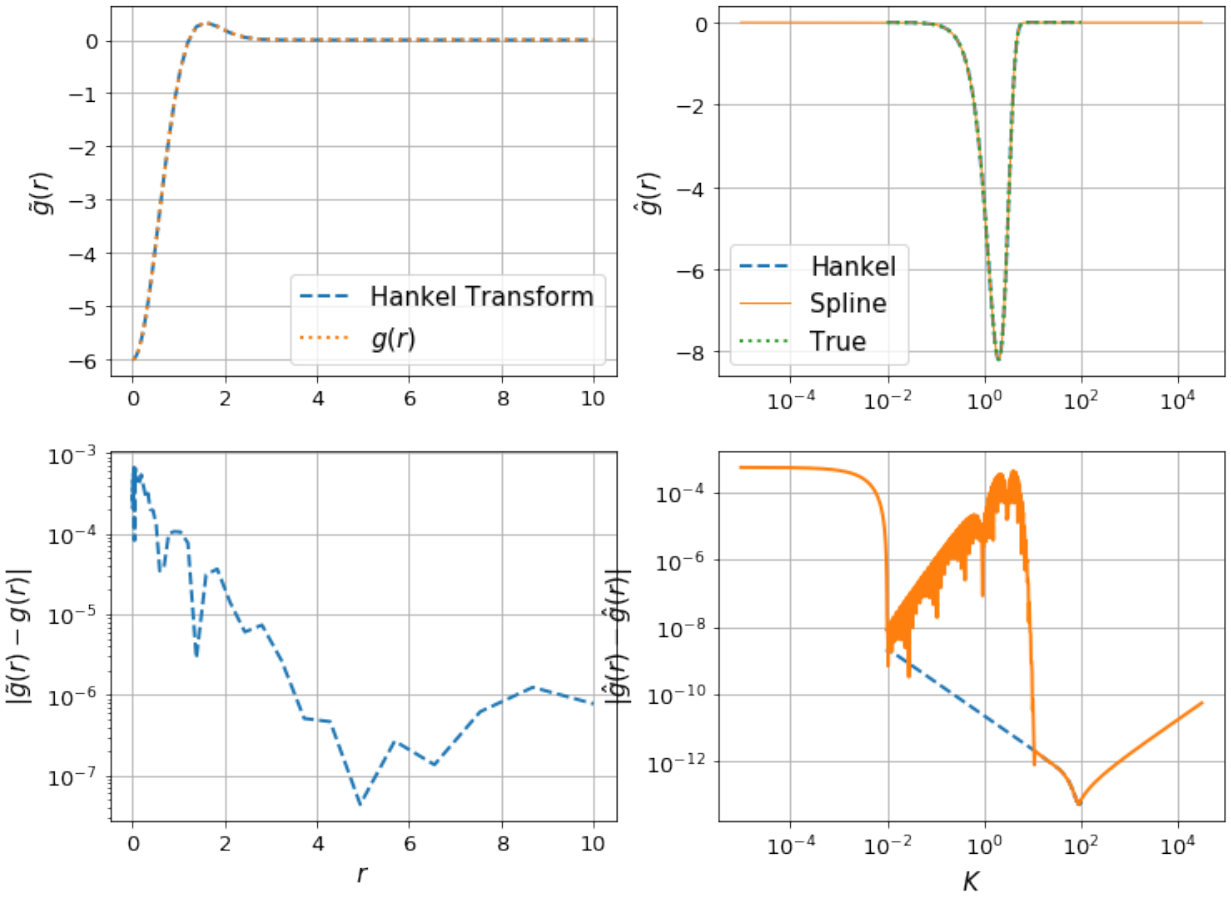
```
In [94]: hback, res, Nback = get_h(ghat_3d, nu=3, K=rfinal[::5], cls=SymmetricFourierTransform, atol=
         hback, Nback, res
```

```
Out[94]: (0.0001953125,
          570,
          array([ -5.99899738e+00,  -5.99590626e+00,  -5.98325134e+00,
                  -5.93166419e+00,  -5.72429124e+00,  -4.93698196e+00,
                  -2.56747300e+00,   2.49885585e-01,   9.44275920e-03,
                   1.07502597e-12]))
```

```
In [87]: sft = SymmetricFourierTransform(3, h=hback, N=Nback)
         K_range = [sft.x.min()/rfinal.max(), sft.x.max()/rfinal.min()]
         K_range
```

```
Out[87]: [9.6382853068197814e-05, 31062.217872340836]
```

```
In [88]: K = np.logspace(-2, 2, 1000)

In [89]: hforward, res, Nforward = get_h(f, nu=3, K=K[::50], cls=SymmetricFourierTransform, atol=1e-8
         hforward, Nforward, res

Out[89]: (0.000125, 1703, array([  5.56817071e+00,   5.56797218e+00,   5.56744663e+00,
                    5.56611565e+00,   5.56276823e+00,   5.55436038e+00,
                    5.53327729e+00,   5.48062292e+00,   5.35044202e+00,
                    5.03663785e+00,   4.32658817e+00,   2.95270658e+00,
                    1.12992711e+00,   1.00971908e-01,   2.32904950e-04,
                    5.45742741e-11,  -5.54278455e-15,  -1.34709891e-15,
                   -3.12381267e-16,  -4.35402965e-17]))

In [91]: make_diagnostic_plots(K, rfinal, ghat_3d, g_3d, f=f,N_forward=Nforward, h_forward=hforward,
```



5.1.4 Testing Forward and Inverse Hankel Transform

This is a simple demo to show how to compute the forward and inverse Hankel transform.

We use the function $f(r) = 1/r$ as an example function. This function is unbounded at $r = 0$ and therefore causes problems with convergence at the origin.

```
In [16]: # Import libraries

         import numpy as np                                    # To define grid
         from hankel import HankelTransform                    # Transforms
```

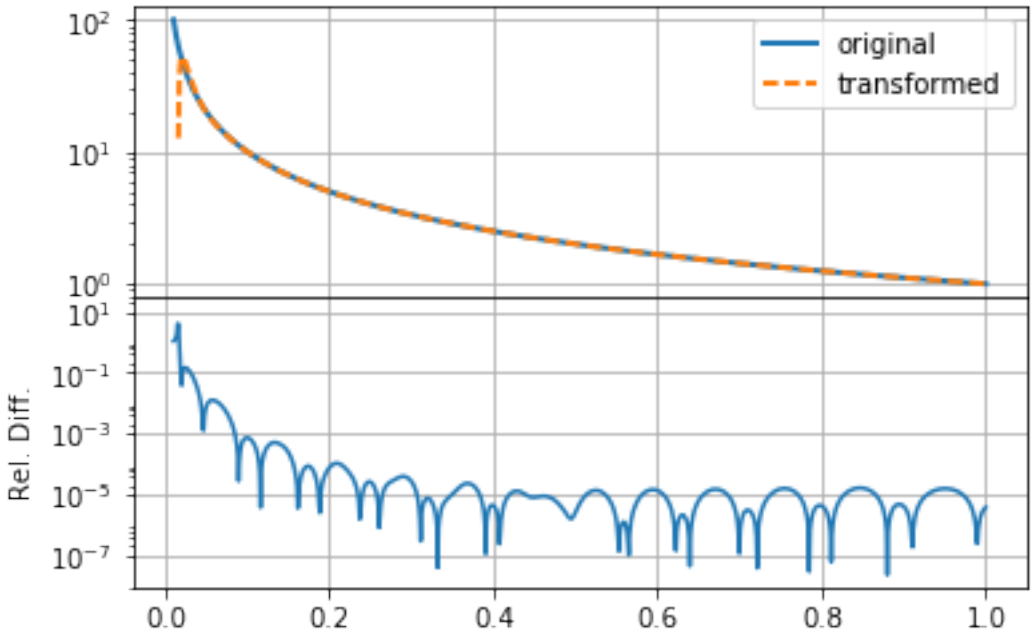
```
from scipy.interpolate import InterpolatedUnivariateSpline as spline    # Spline

import matplotlib.pyplot as plt                                          # Plotting
%matplotlib inline
```

In [2]:
```
# Define grid

r = np.linspace(1e-2,1,1000)                          # Define a physical grid
k = np.logspace(-3,2,100)                             # Define a spectral grid
```

In [3]:
```
# Compute Forward Hankel transform

f    = lambda x : 1/x                                 # Sample Function
h    = HankelTransform(nu=0,N=1000,h=0.005)           # Create the HankelTransform instance, or
hhat = h.transform(f,k,ret_err=False)                 # Return the transform of f at k.
```

In [4]:
```
# Compute Inverse Hankel transform

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

In [15]:
```
# Plot the original function and the transformed functions
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('on')
ax[0].legend(loc='best')
#ax[0].axis('on')

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('on')
ax[1].set_ylabel("Rel. Diff.")
plt.show()
```



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.

5.2 License

5.3 Changelog

5.3.1 v0.3.5 [8 Dec 2017]

Bugfixes - Fixed Python 3 support from v0.3.4

5.3.2 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.

5.3.3 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.

5.3.4 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.

5.3.5 v0.3.1 [5 Jan 2017]

Bugfixes

- Fixed normalisation for inverse transform in `SymmetricFourierTransform`.

Features

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

5.3.6 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.

5.3.7 v0.2.2 [29 April 2016]

Enhancements

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

5.3.8 v0.2.1 [18 Feb 2016]

Bugfixes

- Fixed pip install by changing readme –> README

Enhancements

- updated docs to show dependence on mpmath

5.3.9 v0.2.0 [10 Sep 2014]

Features

- Non-integer orders supported through mpmath.

5.3.10 v0.1.0

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

5.4 API Summary

hankel.hankel	General quadrature method for Hankel transformations.

5.4.1 hankel.hankel

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.

Functions

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

hankel.hankel.get_h

hankel.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=3.2/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.

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.

hankel.hankel.HankelTransform

class hankel.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 nu, 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** : int or 0.5, optional, default = 0

The order of the bessel function (of the first kind) J_nu(x)

N : int, optional, default = 3.2/*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, default = 0.1

The step-size of the integration.

Methods

G(f, h[, k])	The absolute value of the non-oscillatory of the summed series' last term, up to a scaling constant.
__init__([nu, N, h])	
deltaG(f, h, *args, **kwargs)	The slope (up to a constant) of the last term of the series with h
integrate(f[, ret_err, ret_cumsum])	Do the Hankel-type integral of the function f.
transform(f[, k, ret_err, ret_cumsum, inverse])	Do the Hankel-transform of the function f.
xrange([k])	Tuple giving (min,max) x value evaluated by f(x).
xrange_approx(h, nu[, k])	Tuple giving approximate (min,max) x value evaluated by f(x/k).

hankel.hankel.HankelTransform.G

classmethod HankelTransform.**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.

hankel.hankel.HankelTransform.__init__

`HankelTransform.__init__`(*nu=0*, *N=None*, *h=0.05*)

hankel.hankel.HankelTransform.deltaG

classmethod `HankelTransform.deltaG`(*f*, *h*, **args*, ***kwargs*)

The slope (up to a constant) of the last term of the series with h

hankel.hankel.HankelTransform.integrate

`HankelTransform.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

hankel.hankel.HankelTransform.transform

`HankelTransform.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).

hankel.hankel.HankelTransform.xrange

`HankelTransform.`**`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 is a classmethod.

hankel.hankel.HankelTransform.xrange_approx

`classmethod` `HankelTransform.`**`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:

`See` meth:*xrange* (instance method) for the actual x-range under a given choice of parameters.

hankel.hankel.SymmetricFourierTransform

class hankel.hankel.**SymmetricFourierTransform**(*ndim=2*, *a=1*, *b=1*, *N=200*, *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

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

Methods

G(f, h[, k, ndim])	The absolute value of the non-oscillatory of the summed series' last term, up to a scaling constant.
__init__([ndim, a, b, N, h])	
deltaG(f, h, *args, **kwargs)	The slope (up to a constant) of the last term of the series with h

Continued on next page

Table 5.5 – continued from previous page

integrate(f[, ret_err, ret_cumsum])	Do the Hankel-type integral of the function f.
transform(f, k, *args, **kwargs)	Do the *n*-symmetric Fourier transform of the function f.
xrange([k])	Tuple giving (min,max) x value evaluated by f(x).
xrange_approx(h, ndim[, k])	Tuple giving approximate (min,max) x value evaluated by f(x/k).

hankel.hankel.SymmetricFourierTransform.G

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

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.

ndim : float

The number of dimensions of the transform

Returns The value of G.

hankel.hankel.SymmetricFourierTransform.__init__

SymmetricFourierTransform.**__init__**(*ndim=2*, *a=1*, *b=1*, *N=200*, *h=0.05*)

hankel.hankel.SymmetricFourierTransform.deltaG

SymmetricFourierTransform.**deltaG**(*f*, *h*, **args*, ***kwargs*)

The slope (up to a constant) of the last term of the series with h

hankel.hankel.SymmetricFourierTransform.integrate

SymmetricFourierTransform.**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

hankel.hankel.SymmetricFourierTransform.transform

SymmetricFourierTransform.**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.

hankel.hankel.SymmetricFourierTransform.xrange

SymmetricFourierTransform.**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 is a classmethod.

hankel.hankel.SymmetricFourierTransform.xrange_approx

classmethod SymmetricFourierTransform.**xrange_approx**(*h*, *ndim*, *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

ndim : float

Number of dimensions of the transform.

k : array-like, optional

Scales for the transformation. Leave as 1 for an integral.

See also:

See meth:*xrange* (instance method) for the actual x-range under a given choice of parameters.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index

Symbols

D

G

H

I

S

T

X