FiniteDifferences

January 31, 2021

1 Finite Differences

We know that differentiating is an improperly posed problem. However, we might just discretize it, and look at the finite differences which do not care about continuity and so on, they work in discrete spaces. What happens?

Use the forward finite difference

$$\frac{f(x+h)-f(x)}{h} = f'(x) + \frac{h}{2}f''(\xi)$$

Use $f(x) = \sin(x)$ as example, $x = 1$.

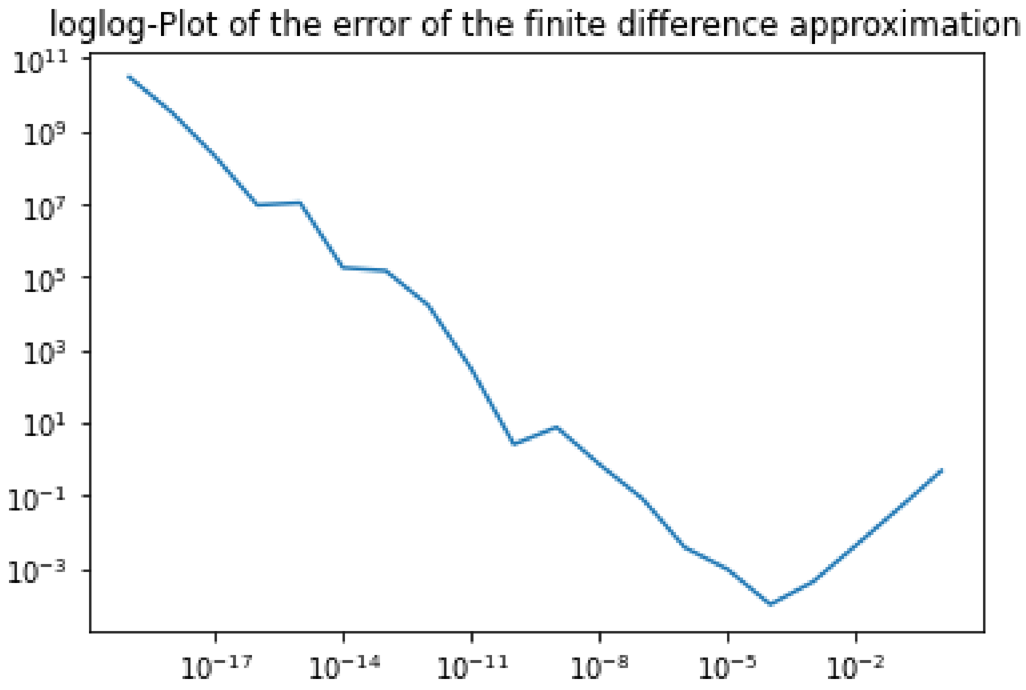
[1]:
```python
import numpy as np
import math
import matplotlib.pyplot as plt
```

[51]:
```python
epsilon=1e-8
def noise():
    return epsilon*(np.random.rand(1)*2-1)

def f(x):
    y=math.sin(x)
    return y+noise()

def fprime(x):
    return math.cos(x)

x=1
```

[52]:
```python
def finite_difference(x,h):
    return (f(x+h)-f(x))/h
```

[53]:
```python
errorlist=np.zeros(20)
hlist=np.zeros(20)
for n in range(0,20):
    h=10**(-n)
    error=abs(finite_difference(x,h)-fprime(x))
    errorlist[n]=error
    hlist[n]=h
```

[54]:
```
plt.loglog(hlist,errorlist)
plt.title('loglog-Plot of the error of the finite difference approximation');
```



When we have an improperly posed problem and disretize it, errors will be huge.

Why, what happens here?

$$\tilde{f}(x) = f(x) + n(x), ||n||_\infty \leq \epsilon$$

If we apply finite differences to $\tilde{f}$, we get

$$|\frac{\tilde{f}(x+h) - \tilde{f}(x)}{h} - f'(x)| = |\frac{h}{2}f''(\xi) + \frac{n(x+h) - n(x)}{h}| \leq \frac{h}{2}||f''||_\infty + 2\frac{\epsilon}{h}$$

For h large, the first term gets (moderately) large.

For h small, the second term gets (very) large.

We call h a regularization parameter: If it is too large, we solve a problem that is far away from the real problem, and the approximation error is high. If it is too small, we solve a problem that is very close to the (unsolvable) real problem, and the result is instable.

What should be used as h here?

The terms should be roughly equal, so the idea might be:

$$|\frac{h}{2}| \sim |\frac{\epsilon}{h}| \Longrightarrow h \sim \sqrt{\epsilon}$$

for our example above, h should be 10^{-4}.

For $\epsilon = 0$ in the program, we still have the machine precision error of the computer, which is roughly 10^{-16} and gives an optimal value of 10^{-8}.

[ ]:

```

```

[ ]:

```

```

Diffusion

January 31, 2021

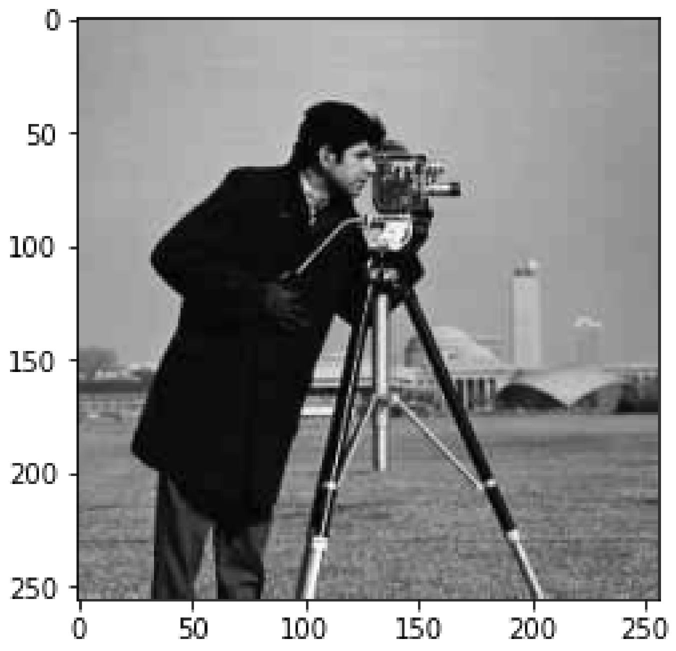
1 Diffusion equation

Take an image as initial value for the diffusion equation

[35]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

[36]:
```
img=mpimg.imread('cameraman.jpg')
plt.imshow(img,cmap='gray')
```

[36]:
```
<matplotlib.image.AxesImage at 0x7f3f1a33dc10>
```



Setup a field for the solution with the initial value.

[37]:
```
T=20
N=256
u=np.zeros([T,N,N])
u[0,:,:]=img
```

Perform finite differences for the diffusion equation

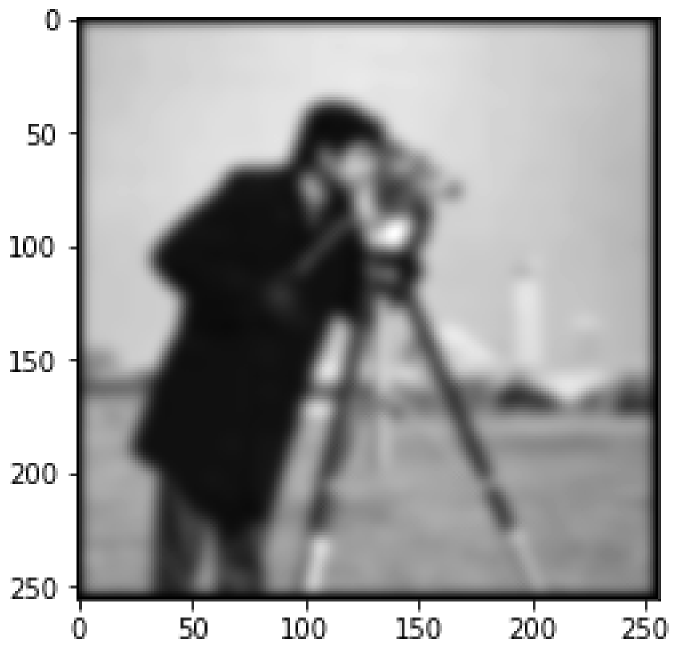
$$u_t = \nabla \cdot (\sigma \nabla u)$$

oder bei konstantem σ

$$u_t = \sigma \Delta u.$$

[40]:
```
dt=0.001
h=1/N
for i in range(0,T-1):
    for k in range(1,N-1):
        for j in range(1,N-1):
            u[i+1,k,j]=u[i,k,j]+dt/
 ↪h*(u[i,k+1,j]+u[i,k-1,j]+u[i,k,j+1]+u[i,k,j-1]-4*u[i,k,j])

finalimg=u[T-1,:,:]
plt.imshow(finalimg,cmap='gray')
```

[40]:
```
<matplotlib.image.AxesImage at 0x7f3f1a591a10>
```



2 Animation

[43]:
```
import matplotlib.animation
from IPython.display import HTML,Image
matplotlib.rc('animation',html='html5')
```

[54]:
```
def ftw_animate(i):
        plt.clf()
        x=np.linspace(0,10,100)
        plt.imshow(u[i,:,:])
        return plt
def film():
    fig=plt.figure()
    ani = matplotlib.animation.FuncAnimation(fig, ftw_animate, frames=19)
    display(HTML(ani.to_jshtml()))
    plt.close(fig)
    return ani
```

[56]:
```
anim=film()
```

```
<IPython.core.display.HTML object>
```

fig=plt.figure() anim = matplotlib.animation.FuncAnimation(fig, animate, frames=100,interval=20) anim

Z=anim.to_html5_video() print(Z[0:200])

anim.save('animation.gif', writer='imagemagick', fps=60)

[ ]:
```
```

SVD

January 31, 2021

Lecture on Inverse Problems, WS 2020/2021, Frank Wübbeling

1 Compare analytical and discrete SVD

We compute the discretized version of the integral operator K that computes the antiderivative (integrating function) on the interval $[0, 1]$, which we already discussed in the lecture. The discretized version is represented by a matrix A.

According to the lecture,

$$Ku(x) = \int_0^1 k(x, y)u(y)\, dy$$

with the kernel function

$$k(x, y) = \begin{cases} 1, & y \leq x \\ 0, & \text{else} \end{cases}.$$

We use the sequence (x_k), $k = 0 \ldots N$, $x_k = k/N$, for discretization.

We discretize using the trapezoidal rule (see lecture on Numerical Analysis):

$$Ku(x_j) = \int_0^1 k(x_j, y)u(y)\, dy \sim h \sum_{l=0}^{N} C_l k(x_j, x_l)u(x_l)$$

with

$$h = 1/N,\, C_l = \begin{cases} \frac{1}{2}, & l = 0 \text{ or } l = N \\ 1, & \text{else} \end{cases}.$$

Setting

$$G, U \in R^{n+1},\, U = (u(x_k)),\, G = ((Ku)(x_k))$$

and

$$A \in R^{(n+1)\times(n+1)}, A = (h\, C_l\, k(x_k, x_l))$$

we get

$$G \sim AU$$

so A is a finited dimensional approximation to K.

We compute the SVD of A and compare with the analytical SVD of K which we computed in the lecture.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
```

Discretize the unit interval. We use $N+1$ equispaced points in $[0,1]$. Discretize the Operator K by a matrix A as above.

[2]:
```
N=128
X=np.linspace(0,1,N+1,endpoint=True)
A=np.zeros([N+1,N+1])
# Discretize using the assembled trapezoidal rule (see num. Analysis)
for i in range(0,N+1):
    for k in range(0,i):
        A[i,k]=1/N
    A[i,0]=A[i,0]*0.5
    A[i,N]=A[i,N]*0.5
```

Compute the SVD using numpy. Note that the diagonal matrix $S = \Sigma$ is in fact returned as a vector. Also note that, different from matlab, not V, but V^t is returned!

Further: S may contain zero elements. This is also different from the definition. All in all, the numpy implementation is simply broken. But we can live with it.

[3]:
```
U,S,V=np.linalg.svd(A)
```

Check whether $A = USV$.

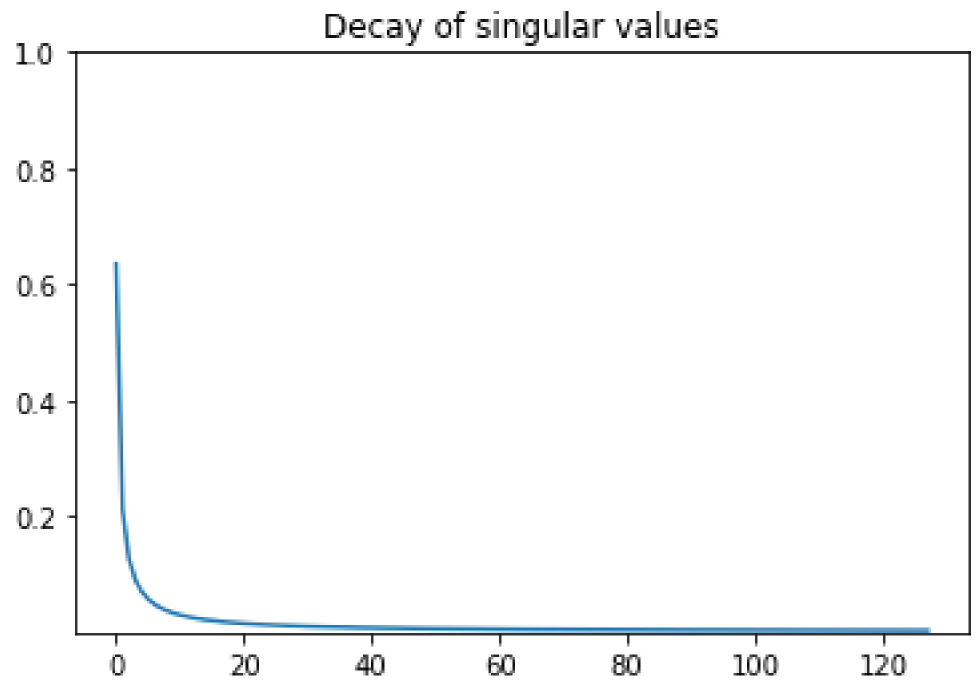
[4]:
```
np.linalg.norm(A-(U*S).dot(V))
```

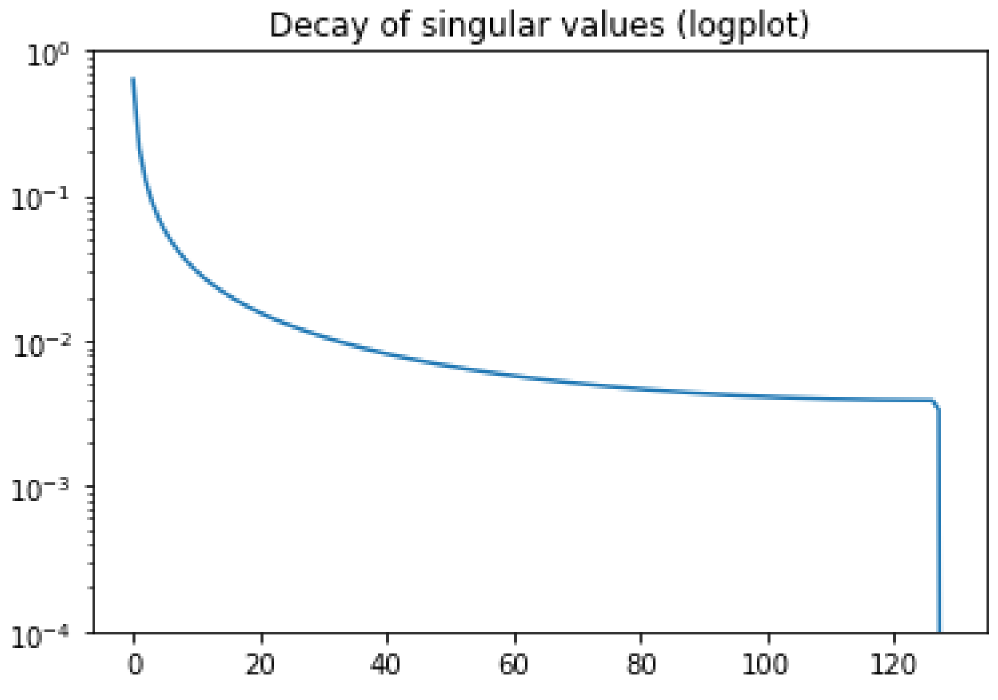
[4]:
```
2.6073465149830414e-15
```

The singular values decay.

[5]:
```
print(max(S),min(S))
plt.plot(S[0:N]);
plt.ylim([1e-4,1])
plt.title('Decay of singular values');
plt.figure();
plt.semilogy(S);
plt.ylim([1e-4,1])
plt.title('Decay of singular values (logplot)');
```

```
0.6353803652022266 5.735074912535327e-17
```

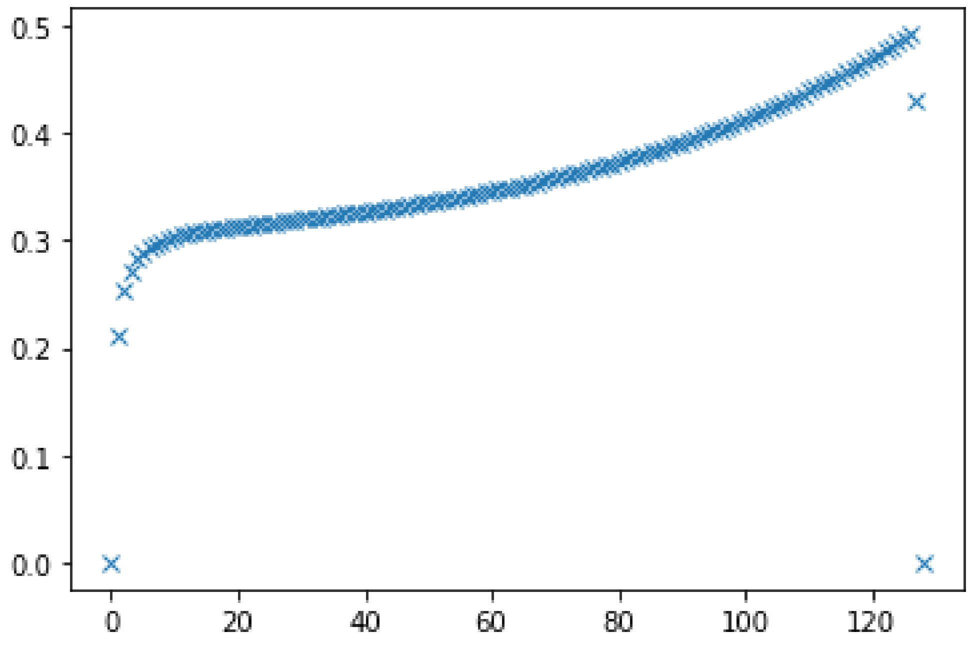




We computed the singular value $\sigma_k = O(\frac{1}{k})$, so $k\sigma_k$ should be more or less constant.

```
[6]: plt.plot((np.arange(0,N+1))*S,'x')
```

```
[6]: [<matplotlib.lines.Line2D at 0x7fed34392590>]
```



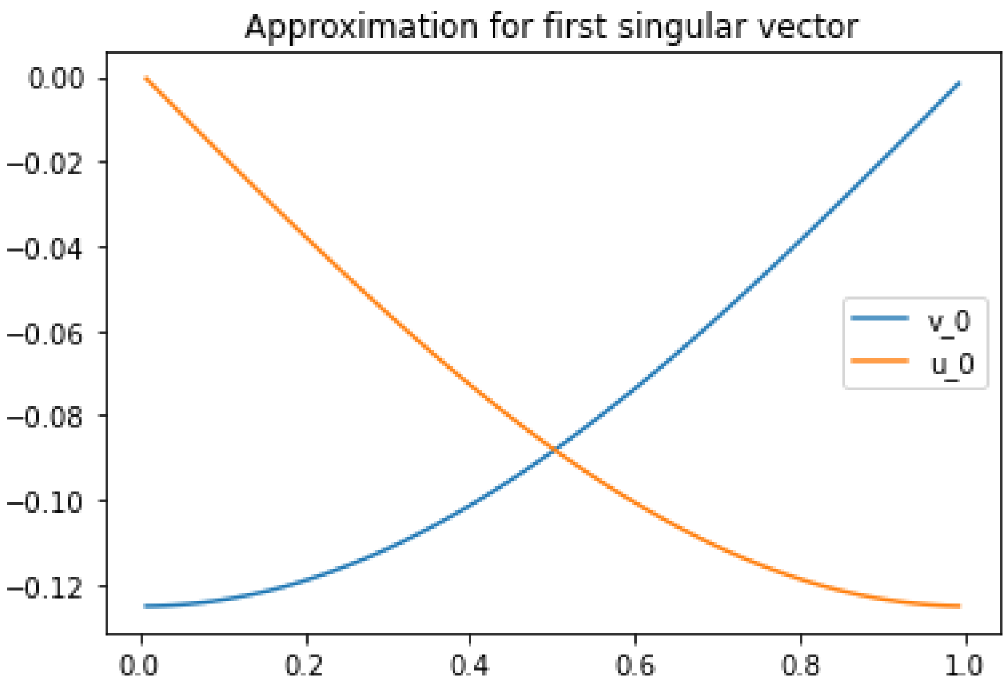
In fact, this approximation is acceptable, but not too good. There must still something be going on.

However, the approximation of the singular vectors is excellent, this is more or less exactly what we computed in the lecture.

Note that singular vectors can only be unique up to a factor with absolute value 1. Here, numpy computes the negative of the functions in the lecture.

```
[7]: plt.plot(X[1:N],V[0,1:N],X[1:N],U[1:N,0])
     plt.legend(['v_0','u_0'])
     plt.title('Approximation for first singular vector')
```

```
[7]: Text(0.5, 1.0, 'Approximation for first singular vector')
```



2 Testing discretizations

We want to show that we were very lucky that the result above was ok. We compare two different discretizations of the integral operator, Trapezoidal Rule and Simpson Rule.

We solve the inverse problem $Kf = g$ using both discretizations. Everything else is as above.

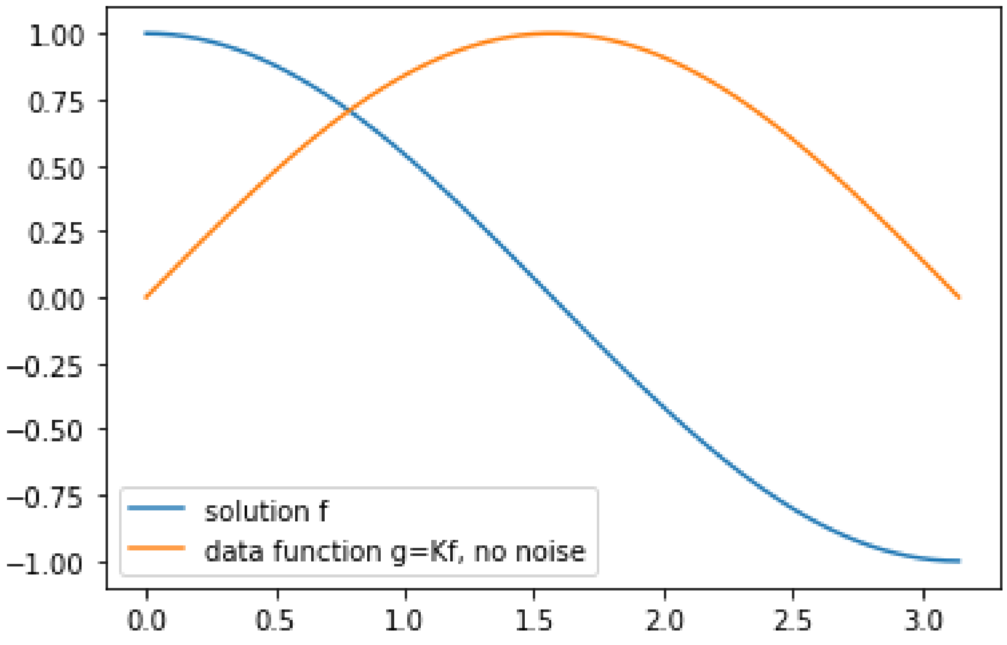
[8]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
```

As a test problem, we use $Af = g = sin$ with the exact solution $f = \cos$. As above, we compute discretized versions F and G on equidistant points.

[9]:
```
def f(x):
    return np.cos(x)

def g(x):
    return np.sin(x)

N=128
X=np.linspace(0,math.pi,N+1,endpoint=True)

F=f(X)
```

```
G=g(X)
plt.plot(X,F,X,G)
plt.legend(['solution f','data function g=Kf, no noise'])
```

```
[9]: <matplotlib.legend.Legend at 0x7fed3429f650>
```



This time, we use only the kernel function for A and write h and the coefficients D for the trapezoidal/Simpson rule explicitly, as in the lecture.

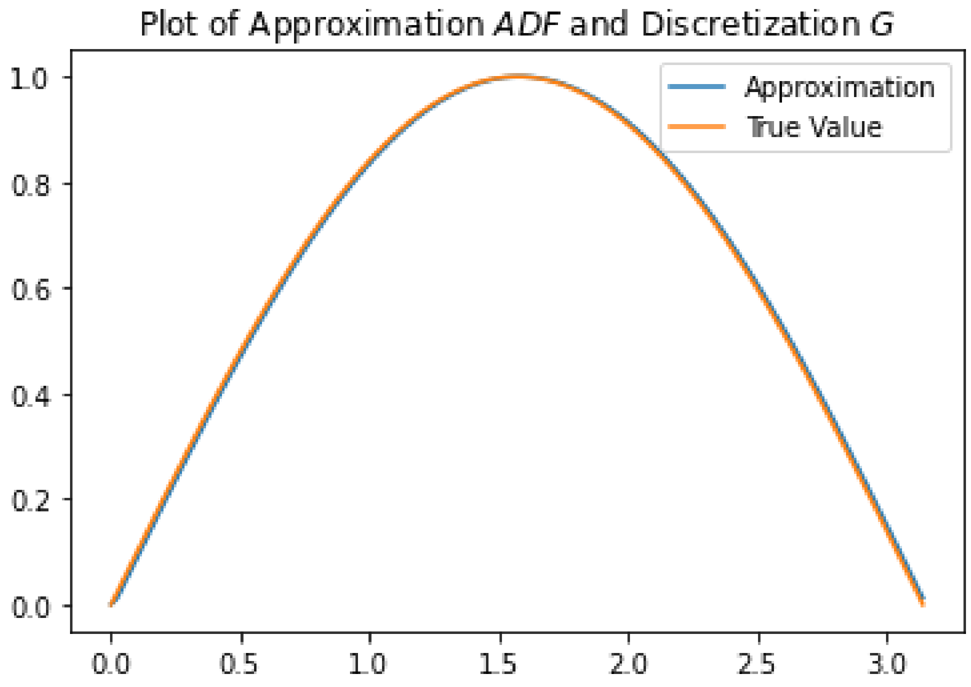
```
[10]: A=np.zeros([N+1,N+1])
      for i in range(0,N+1):
          for k in range(0,i):
              A[i,k]=1
```

```
[11]: D=np.ones(N-1);
      D=np.append(D,0.5);
      D=np.insert(D,0,0.5);
      #Simpson
      #D[0]=1
      #D[range(1,N,2)]=4
      #D[range(2,N,2)]=2
      #D[N]=1
      #D=D/3
      D=D*math.pi/N;
      D=np.diag(D)
```

We check that ADF is a good approximation for G.

```
[12]: GSchlange=A.dot(D.dot(F))
      plt.plot(X,GSchlange,X,G)
      plt.legend(['Approximation','True Value'])
      plt.title('Plot of Approximation $ADF$ and Discretization $G$')
```

```
[12]: Text(0.5, 1.0, 'Plot of Approximation $ADF$ and Discretization $G$')
```



Compute the SVD and check that it is correct. numpy computes a zero singular value, we throw it out.

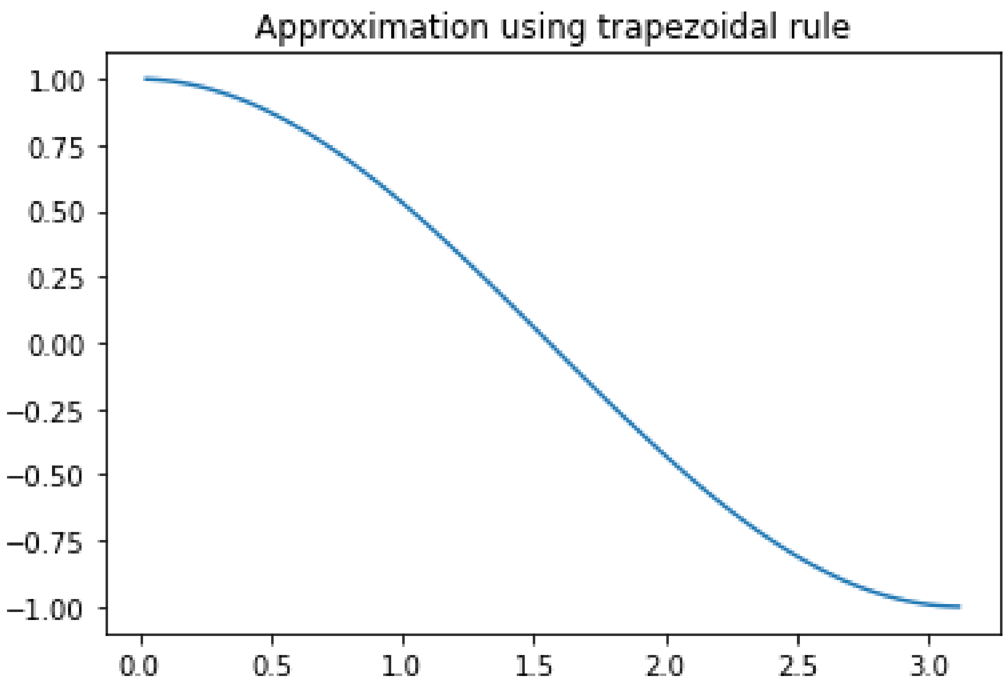
```
[13]: U,S,V=np.linalg.svd(A.dot(D));
      S=S[0:N]
      U=U[:,0:N]
      V=V[0:N,:]
      np.linalg.norm(A.dot(D)-(U*S).dot(V))
```

```
[13]: 8.508278869094I9e-15
```

Compute the Minimum Norm Solution using the SVD.

```
[14]: B=(V.transpose()*(1/S)).dot(U.transpose())
      Fschlange=B.dot(G)
      plt.plot(X[1:N],Fschlange[1:N])
plt.title('Approximation using trapezoidal rule')
```

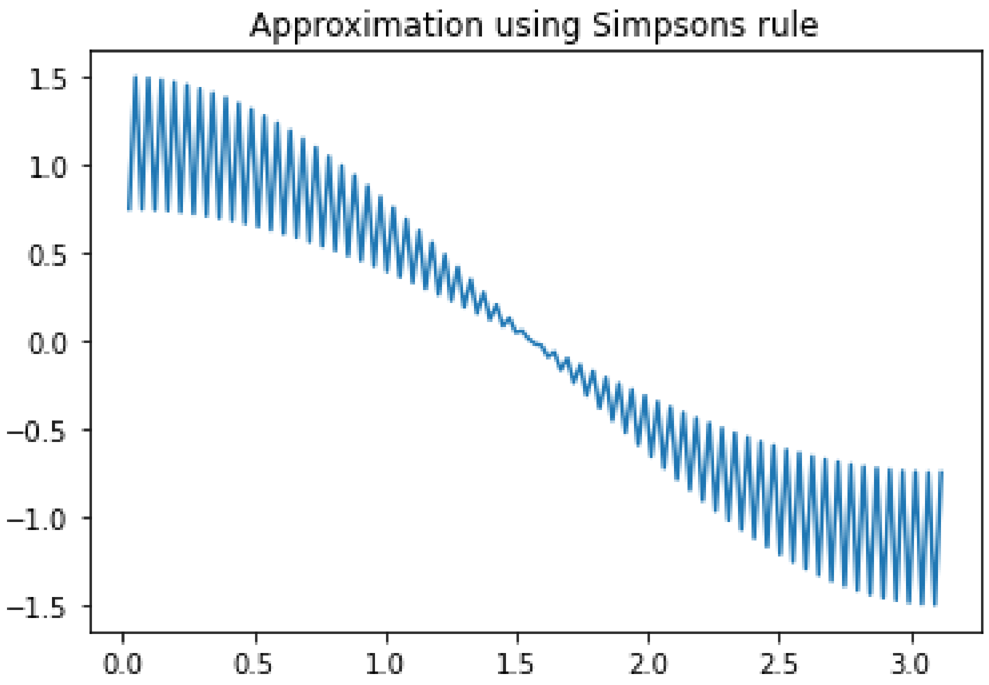
```
[14]: Text(0.5, 1.0, 'Approximation using trapezoidal rule')
```



Now we run exactly the same thing for Simpson's rule.

```
[15]: D=np.ones(N-1);
D=np.append(D,0.5);
D=np.insert(D,0,0.5);
#Simpson
D[0]=1
D[range(1,N,2)]=4
D[range(2,N,2)]=2
D[N]=1
D=D/3
D=D*math.pi/N;
D=np.diag(D)
U,S,V=np.linalg.svd(A.dot(D));
S=S[0:N]
U=U[:,0:N]
V=V[0:N,:]
np.linalg.norm(A.dot(D)-(U*S).dot(V))
B=(V.transpose()*(1/S)).dot(U.transpose())
Fschlange=B.dot(G)
plt.plot(X[1:N],Fschlange[1:N])
plt.title('Approximation using Simpsons rule')
```

```
[15]: Text(0.5, 1.0, 'Approximation using Simpsons rule')
```



And obviously, this time it just did not work. We get no approximation to the true solution. This does not change with N. Try it.

Essence: Discretizing inverse problems with point evaluations is hazardous. Better use moment methods.

```
[ ]:
```

TV Shepp Logan

January 31, 2021

1 Denoising and Restoration with Tikhonov and TV

We use the pylops-package for this. This is solely based on their example code. See Ravasi, M., and Vasconcelos I., PyLops–A linear-operator Python library for scalable algebra and optimization, Software X, (2020) and https://pylops.readthedocs.io/en/latest/index.html.

(Almost) all linear regularization methods for compact operators can be represented as

$$K_\alpha^+ g = \sum_k g_\alpha(\sigma_k)(g, v_k)u_k$$

for a singular system (σ_k, u_k, v_k). We proved the regularization property assuming (among others)

$$\forall_{\alpha,\sigma} \sigma g_\alpha(\sigma) \leq C,\, g_\alpha(\sigma) \leq C_\alpha, g_\alpha(\sigma) \to_{\alpha\to 0} \frac{1}{\sigma}.$$

We consider the inverse problem $Ku = g$, $||g_\delta - g|| \leq \delta$, $g \in D(K^+)$. We choose a regularization parameter $\alpha = \alpha(\delta, g_\delta)$ and compute the regularized solution $u_\alpha^+ = K_\alpha^+ g_\delta$.

Let us assume for a moment that the measurement is exact, but we don't know, so $g = g_\delta$ but $\delta > 0$, so we choose $\alpha > 0$. Then we have for the minimum norm solution u^+

$$||u_\alpha^+ - u^+||^2 = ||K_\alpha^+ g - K^+ g||^2 = \sum_k (g_\alpha(\sigma_k) - \frac{1}{\sigma_k}(g, v_k))^2$$

So even in the case of exact data, we will get an error now. We have $g_\alpha(\sigma_k) \sim \frac{1}{\sigma}$ provided α is small enough for a fixed σ. On the other hand, for fixed α, $\sigma g_\alpha(\sigma) \to_{\sigma\to 0} 0$.

Comparing the terms in both series, this means that terms for small k (or large σ_k) will be almost the same, while for large k (or small σ_k) the regularized terms are (much) smaller in absolute value, they are damped.

Going with our assumption that the singular vectors are highly oscillating for k large, we take away these vectors from the solution. If the vectors are even sine/cosine, this amounts to taking the Fourier transform (representing the function in sine/cosine) and throwing away the high coefficients. Effectively, we are smoothing the function.

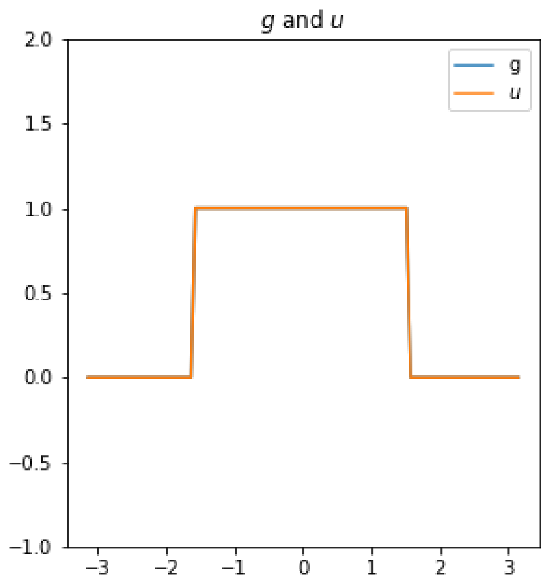
For our numerical experiment, we choose $K = I$ (which is not a compact operator). We choose $u_k(x) := v_k(x) := \frac{1}{2\pi}e^{ikx}$, $\sigma_k := 1$, which fits into our regularization scheme for the SVD.

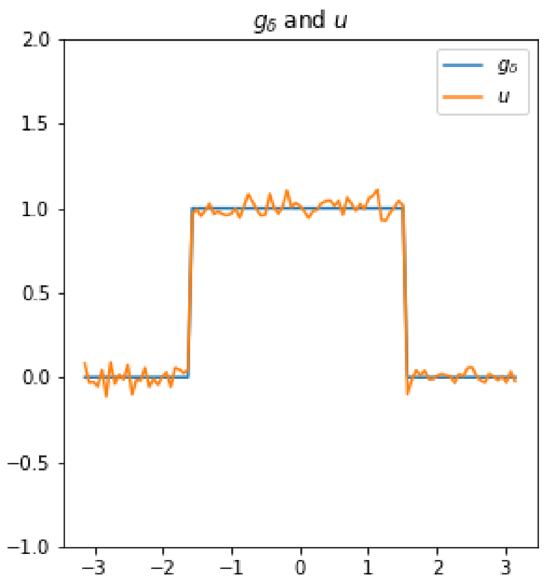
We discretize the problem with a stepsize of $1/N$. Choosing trapezoidal rule for the scalar products then yields the discrete Fourier Transform.

[1]:
```python
#!pip install --user pylops
import numpy as np
import matplotlib.pyplot as plt
import pylops
import math
plt.close('all')
np.random.seed(1)
```

We start with a 1D example that shows the effect. We choose u as the characteristic function of $[0.5 - p, 0.5 + p]$ and assume that our measurement is exact, but we don't know so we regularize with an $\alpha(\delta)$. We also look at a second noisy measurement.

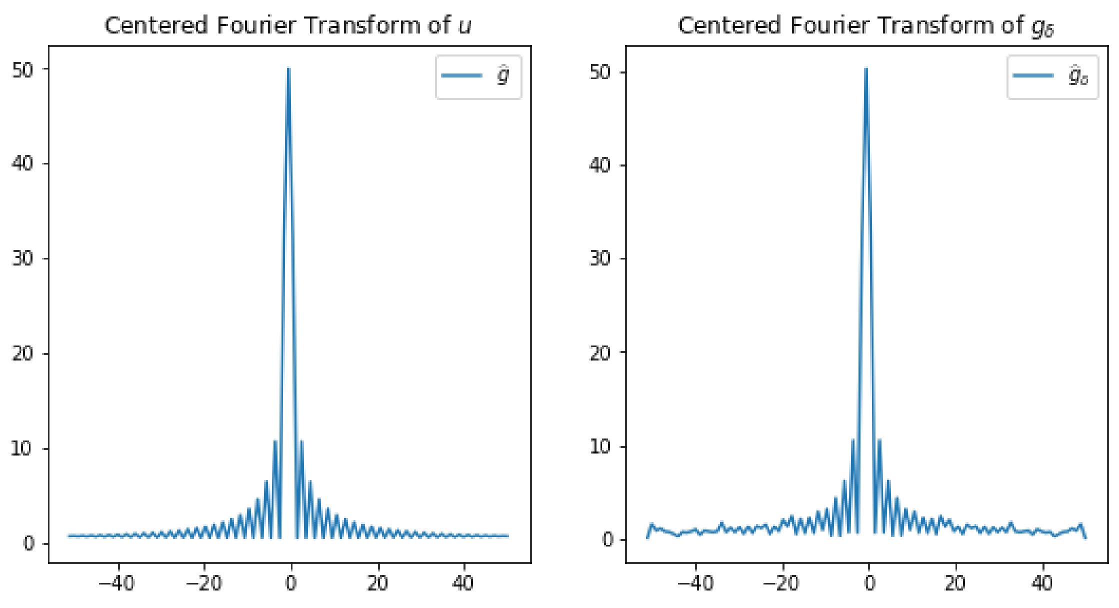
[2]:
```python
N = 101
X=np.linspace(-math.pi,math.pi,N,endpoint=True)
u = np.zeros(N)
p = N//4
u[N//2-p:N//2+p] = 1

# g has no noise. gdelta is noisy.

dev=0.05
n = np.random.normal(0, dev, N)
g = u
gdelta=u+n

plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.plot(X,u,X,g)
plt.legend(['g','$u$'])
plt.title('$g$ and $u$')
plt.ylim([-1,2])
plt.subplot(122)
plt.plot(X,u,X,gdelta)
plt.legend(['$g_\delta$','$u$'])
plt.title('$g_\delta$ and $u$')
plt.ylim([-1,2]);
```





We first compute the Fourier Transform of u and check whether it decays.

[3]:
```
F=np.fft.fft(u)
G=np.fft.fft(g)
Gdelta=np.fft.fft(gdelta)
FourX=np.linspace(-N//2,N//2,N,endpoint=True)
plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.plot(FourX,np.fft.fftshift(np.abs(F)))
plt.legend(['$\widehat g$'])
plt.title('Centered Fourier Transform of $u$')
plt.subplot(122)
plt.plot(FourX,np.fft.fftshift(np.abs(Gdelta)))
plt.title('Centered Fourier Transform of $g_\delta$')
plt.legend(['$\widehat g_\delta$']);
#plt.ylim([-1,2]);
```



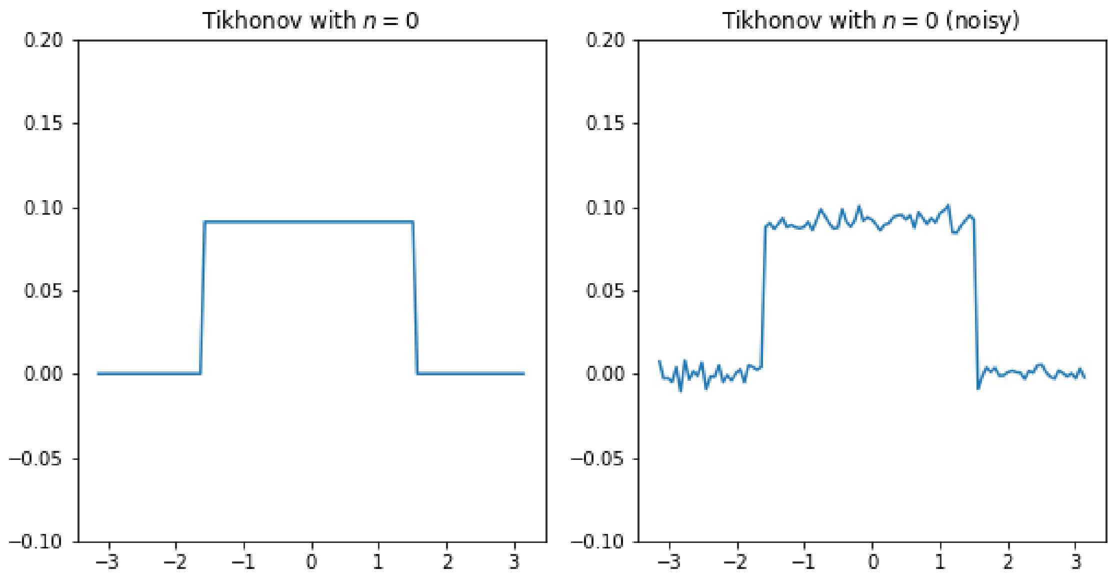
We note that $\widehat{g}$ decays fast, even if it does not fit our assumption (g is differentiable). Also note that we number the Fourier coefficients from -N/2 to N/2, and plot them accordingly (a_0 is in the center).

For the noisy measurement, coefficients with small $|k|$ are ok, for large k they are heavily distorted (relative).

We start with Tikhonov Regularization and investigate the effect of different choices of α.

[4]:
```
Iop=pylops.Identity(N)

D2op = pylops.SecondDerivative(N, edge=True)
D1op = pylops.FirstDerivative(N, edge=True)
D0op = pylops.Identity(N)

lamda = 20

xinv0 = pylops.optimization.leastsquares.RegularizedInversion(Iop, [D0op], g,
                                    epsRs=[np.sqrt(lamda/2)],**dict(iter_lim=30))
xinv1 = pylops.optimization.leastsquares.RegularizedInversion(Iop, [D1op], g,
                                    epsRs=[np.sqrt(lamda/2)],**dict(iter_lim=30))
xinv2 = pylops.optimization.leastsquares.RegularizedInversion(Iop, [D2op], g,
                                    epsRs=[np.sqrt(lamda/2)],**dict(iter_lim=30))
xinv0delta = pylops.optimization.leastsquares.RegularizedInversion(Iop, [D0op],␣
 ↪gdelta,
                                    epsRs=[np.sqrt(lamda/2)],**dict(iter_lim=30))
xinv1delta = pylops.optimization.leastsquares.RegularizedInversion(Iop, [D1op],␣
 ↪gdelta,
                                                epsRs=[np.sqrt(lamda/2)],**dict(iter_lim=30))
xinv2delta = pylops.optimization.leastsquares.RegularizedInversion(Iop, [D2op],␣
 ↪gdelta,
                                                epsRs=[np.sqrt(lamda/2)],**dict(iter_lim=30))

plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.plot(X,xinv0)
#plt.legend(['$u_\alpha^+$'])
plt.title('Tikhonov with $n=0$')
plt.ylim([-0.1,0.2])
plt.subplot(122)
plt.plot(X,xinv0delta)
plt.title('Tikhonov with $n=0$ (noisy)')
plt.ylim([-0.1,0.2])
```

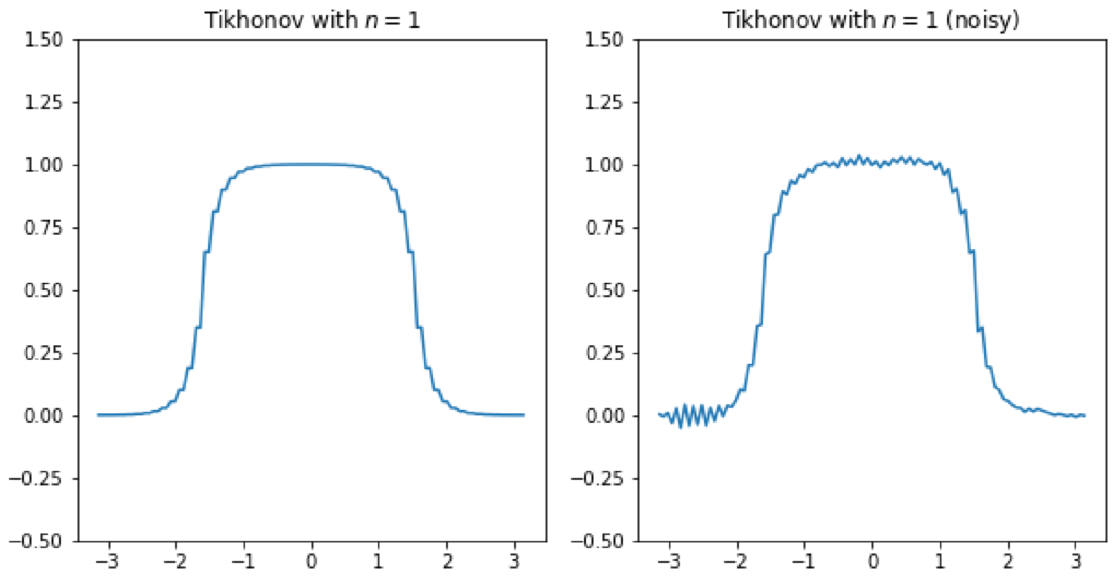
[4]:
```
(-0.1, 0.2)
```



Choosing $n = 0$ (true Tikhonov in the L^2-norm), just scales the input and does not get rid of the noise. Now choose $n = 1$.

[5]:
```
plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.plot(X,xinv1)
#plt.legend(['$u_\alpha^+$'])
plt.title('Tikhonov with $n=1$')
plt.ylim([-0.5,1.5])
plt.subplot(122)
plt.plot(X,xinv1delta)
plt.title('Tikhonov with $n=1$ (noisy)')
plt.ylim([-0.5,1.5])
```

```
[5]: (-0.5, 1.5)
```

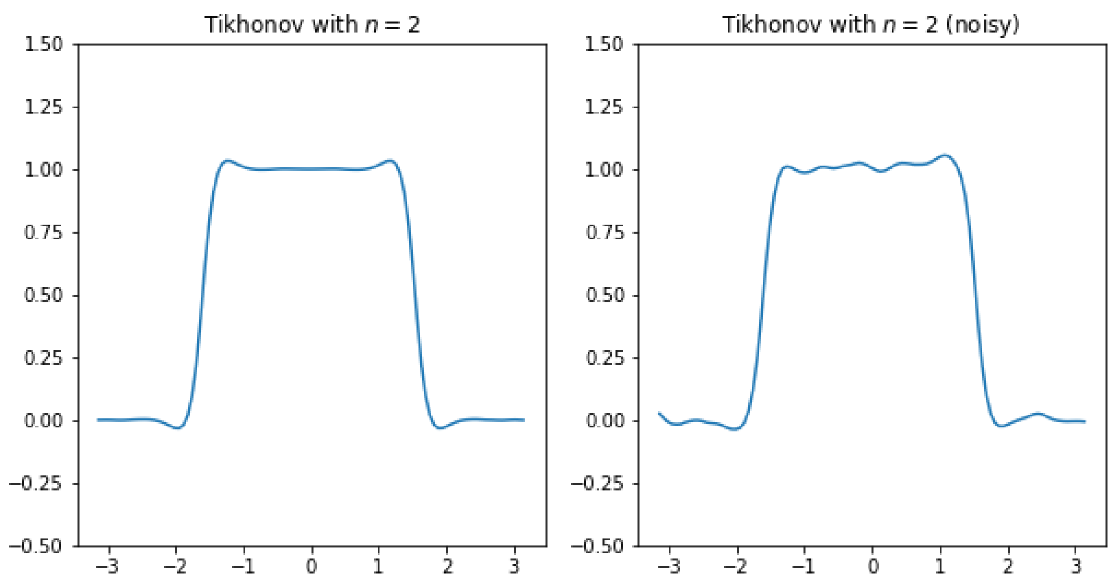


We note: Now we are removing the noise, but the figure is far too smooth.

Reason: To represent a discontinuity, we need higher order oscillating functions. Since linear regularization will always damp these, the resulting approximation will always be smooth.

We test for $n = 2$.

```
[6]: plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.plot(X,xinv2)
#plt.legend(['$u_\alpha^+$'])
plt.title('Tikhonov with $n=2$')
plt.ylim([-0.5,1.5])
plt.subplot(122)
plt.plot(X,xinv2delta)
plt.title('Tikhonov with $n=2$ (noisy)')
plt.ylim([-0.5,1.5]);
```

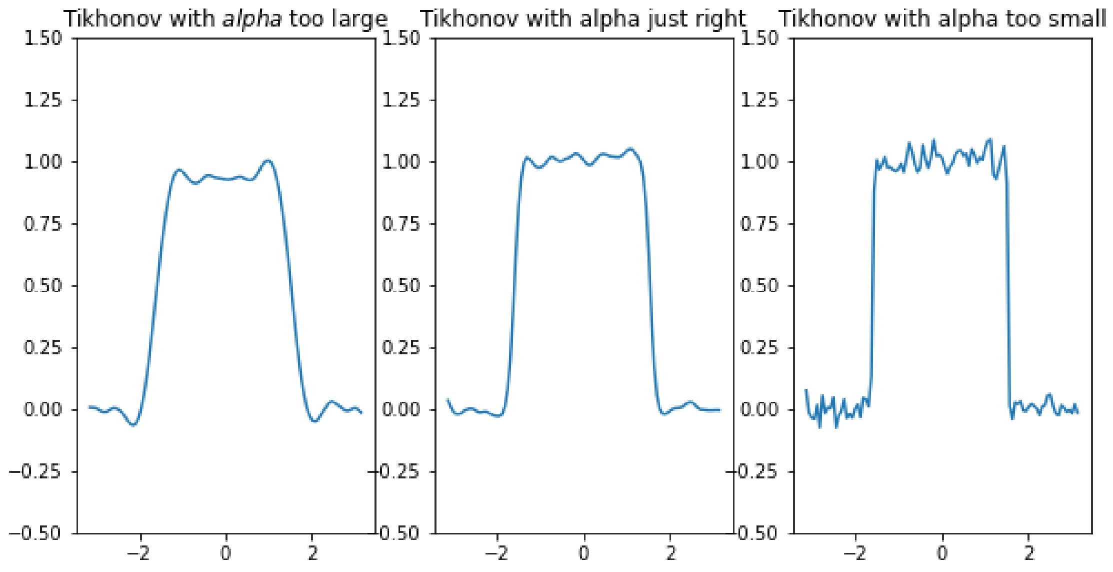


Keeps the discontinuity better, but does not completely remove the noise.

Conclusion: Choosing the right norm (like choosing the correct regularization parameter) is a delicate task.

So we turn to choosing the right regularization parameter. We repat our experiment for $n = 2$ with different values of α.

[7]:
```
lamda = 10
p=10
xinv0 = pylops.optimization.leastsquares.RegularizedInversion(Iop, [D2op],␣
 ↪gdelta,
                                    epsRs=[np.sqrt(lamda/
 ↪2)*p],**dict(iter_lim=30))
xinv1 = pylops.optimization.leastsquares.RegularizedInversion(Iop, [D2op],␣
 ↪gdelta,
                                    epsRs=[np.sqrt(lamda/2)],**dict(iter_lim=30))
xinv2 = pylops.optimization.leastsquares.RegularizedInversion(Iop, [D2op],␣
 ↪gdelta,
                                    epsRs=[np.sqrt(lamda/2)/
 ↪p],**dict(iter_lim=30))
plt.figure(figsize=(10, 5))
plt.subplot(131)
plt.plot(X,xinv0)
#plt.legend(['$u_\alpha^+$'])
plt.title('Tikhonov with $alpha$ too large')
plt.ylim([-0.5,1.5])
plt.subplot(132)
plt.plot(X,xinv1)
plt.title('Tikhonov with alpha just right')
plt.ylim([-0.5,1.5]);
plt.subplot(133)
plt.plot(X,xinv2)
plt.title('Tikhonov with alpha too small')
plt.ylim([-0.5,1.5]);
```



2 Nonlinear Regularization

Our goal is therefore to find a regularization that does not penalize jumps.

In TV (total variation) regularization, the regularized solution is defined as

$$u_\alpha^+ = \arg\min_u ||Ku - g||^2 + ||u||_{TV}, \; ||u||_{TV} := ||\nabla u||_1$$

(for differentiable u and extended to a subset of L^2 including step functions). Why does the 1-Norm make a difference?

Assume $u : [-1,1] \mapsto R$, u differentiable, u monotonous. Then obviously

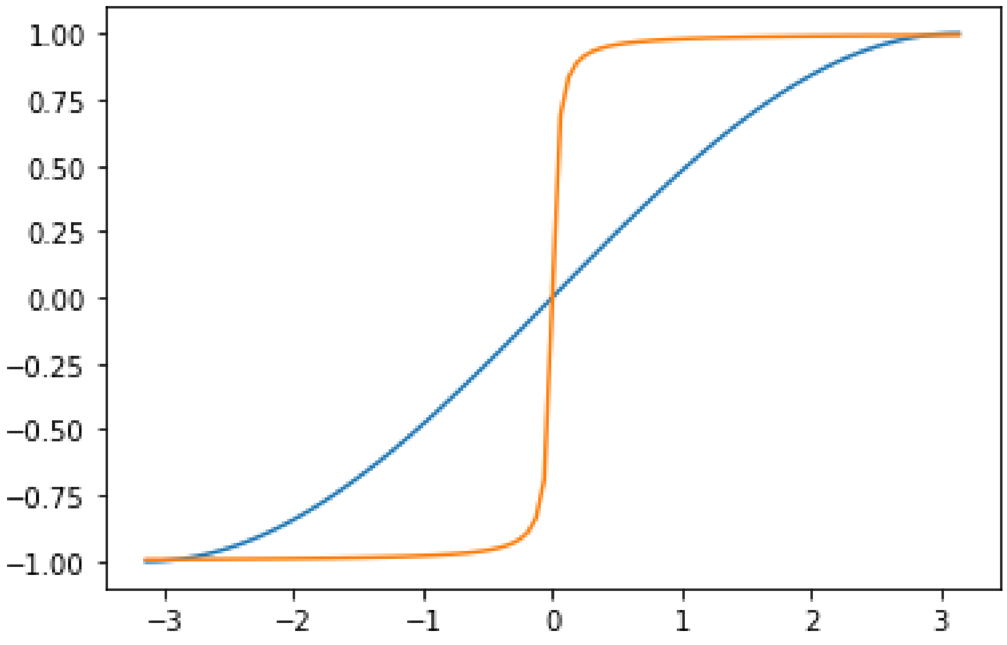
$$||u||_{TV} = \int_{-1}^{1} |u'(x)|\, dx = \pm \int_{-1}^{1} u'(x)\, dx = |u(1) - u(-1)|$$

which says that any such function from $(-1, u(-1))$ to $(1, u(1))$ has the same (Semi-) Norm. So it does not penalize the shape of a monotonous function, particularly it does not penalize edges. Rather, it penalizes non-monotonicity (and thus the size of jumps).

Simple example: the two functions below share the same TV-seminorm, but the 2-norm of the derivative of the step-function is very much larger, that's why the 2-norm will favor a smooth curve.

[8]:
```
X1=np.linspace(-1,1,N)
u1=np.sin(X1*math.pi/2)
u2=np.arctan(30*math.pi*X1)/math.pi*2
plt.plot(X,u1,X,u2)
print(np.linalg.norm(D1op*u1),np.linalg.norm(D1op*u2))
```

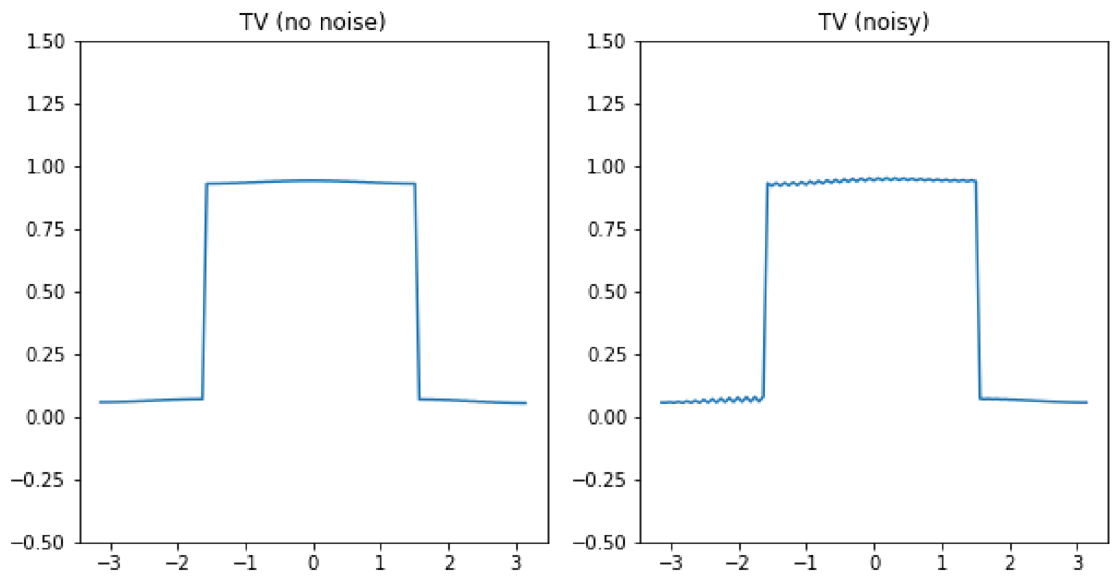
```
0.2221087036983384 0.9213424954358104
```



Now back to our example. This time, there is no analytical representation of the result, so we must resort to numerical optimization.

[9]:
```
mu = 0.1
lamda = 0.4
niter_out = 50
niter_in = 3

xinv0, niter = pylops.optimization.sparsity.SplitBregman(Iop, [D1op], g,␣
 ↪niter_out,
                                                        niter_in, mu=mu, epsRL1s=[lamda],
                                                        tol=1e-4, tau=1.,
                                                        **dict(iter_lim=30, damp=1e-10))
xinv1, niter = pylops.optimization.sparsity.SplitBregman(Iop, [D1op], gdelta,␣
 ↪niter_out,
                                                 niter_in, mu=mu, epsRL1s=[lamda],
                                                 tol=1e-4, tau=1.,
                                                 **dict(iter_lim=30, damp=1e-10))


plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.plot(X,xinv0)
#plt.legend(['$u_\alpha^+$'])
plt.title('TV (no noise)')
plt.ylim([-0.5,1.5])
plt.subplot(122)
plt.plot(X,xinv1)
plt.title('TV (noisy)')
plt.ylim([-0.5,1.5])
```

[9]:
```
(-0.5, 1.5)
```

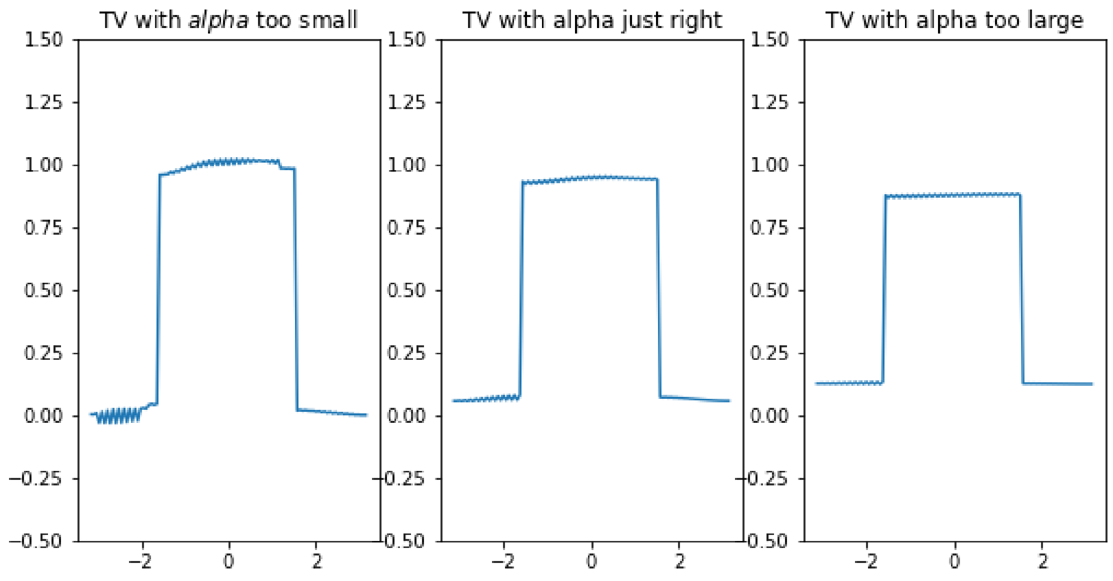


TV does a pretty good job at keeping the discontinuity.

Again, we check what happens if α is not chosen correctly.

[10]:
```
p=8
xinv0, niter = pylops.optimization.sparsity.SplitBregman(Iop, [D1op], gdelta,␣
 ↪niter_out,
                                                 niter_in, mu=mu*p, epsRL1s=[lamda],
                                                 tol=1e-4, tau=1.,
                                                 **dict(iter_lim=30, damp=1e-10))
```

```
xinv2, niter = pylops.optimization.sparsity.SplitBregman(Iop, [D1op], gdelta,␣
 ↪niter_out,
                                                          niter_in, mu=mu/2, epsRL1s=[lamda],
                                                          tol=1e-4, tau=1.,
                                                          **dict(iter_lim=30, damp=1e-10))

plt.figure(figsize=(10, 5))
plt.subplot(131)
plt.plot(X,xinv0)
#plt.legend(['$u_\alpha^+$'])
plt.title('TV with $alpha$ too small')
plt.ylim([-0.5,1.5])
plt.subplot(132)
plt.plot(X,xinv1)
plt.title('TV with alpha just right')
plt.ylim([-0.5,1.5]);
plt.subplot(133)
plt.plot(X,xinv2)
plt.title('TV with alpha too large')
plt.ylim([-0.5,1.5]);
```

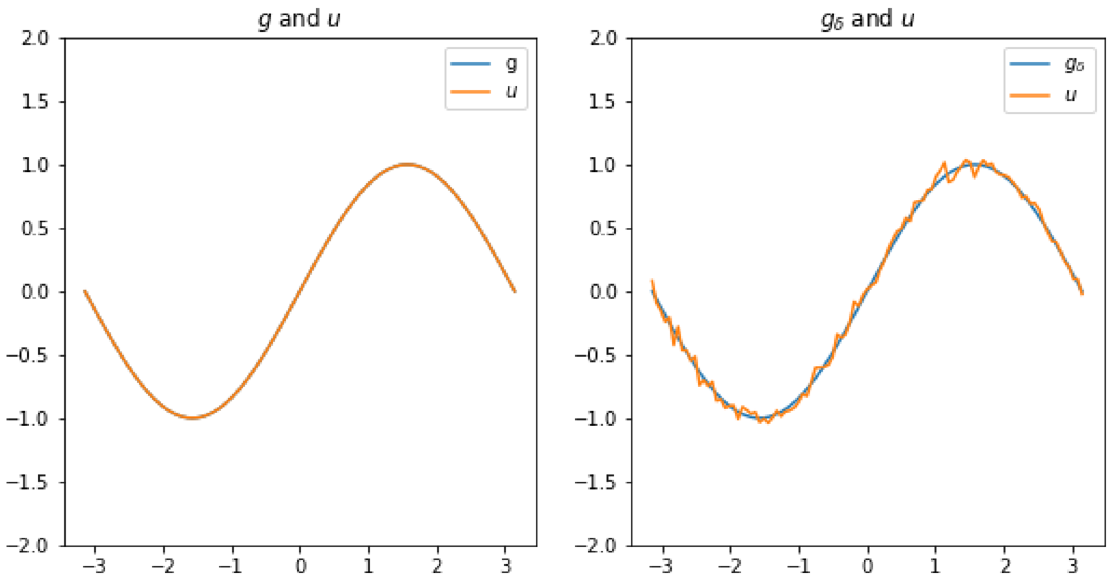


Although choosing α too large or too small, the result is phantastic: We are losing contrast, but generally, but the overall shape (which is important in imaging) stays the same.

However, there is a drawback. Consider now a function which is not so suited for TV, the sine.

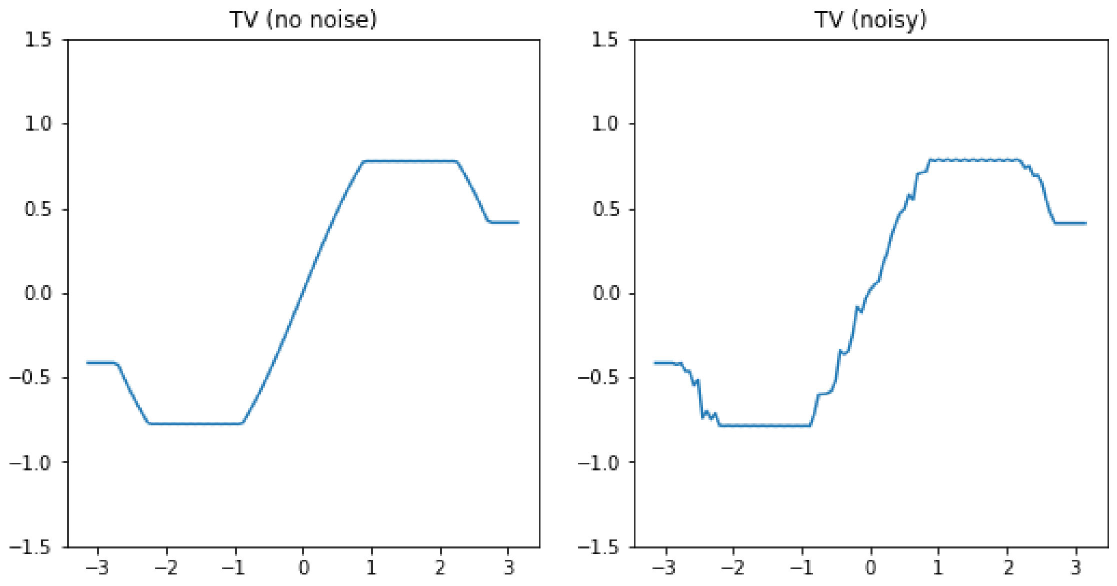
[11]:
```
u=np.sin(X)
g = u
```

```
gdelta=u+n

plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.plot(X,u,X,g)
plt.legend(['g','$u$'])
plt.title('$g$ and $u$')
plt.ylim([-2,2])
plt.subplot(122)
plt.plot(X,u,X,gdelta)
plt.legend(['$g_\delta$','$u$'])
plt.title('$g_\delta$ and $u$')
plt.ylim([-2,2]);
```



[12]:
```
mu = 0.1
lamda = 0.4
niter_out = 50
niter_in = 3

xinv0, niter = pylops.optimization.sparsity.SplitBregman(Iop, [D1op], g,␣
 ↪niter_out,
                                                          niter_in, mu=mu, epsRL1s=[lamda],
                                                          tol=1e-4, tau=1.,
                                                          **dict(iter_lim=30, damp=1e-10))
xinv1, niter = pylops.optimization.sparsity.SplitBregman(Iop, [D1op], gdelta,␣
 ↪niter_out,
                                                          niter_in, mu=mu, epsRL1s=[lamda],
                                              tol=1e-4, tau=1.,
                                              **dict(iter_lim=30, damp=1e-10))


plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.plot(X,xinv0)
#plt.legend(['$u_\alpha^+$'])
plt.title('TV (no noise)')
plt.ylim([-1.5,1.5])
plt.subplot(122)
plt.plot(X,xinv1)
plt.title('TV (noisy)')
plt.ylim([-1.5,1.5])
```

[12]:
```
(-1.5, 1.5)
```



Smooth transitions turn to plateaus, or spots in images. For medical imaging, the doctor will have to decide whether this is an artefact or whether this is real.

He will usually rather keep the noisy image.

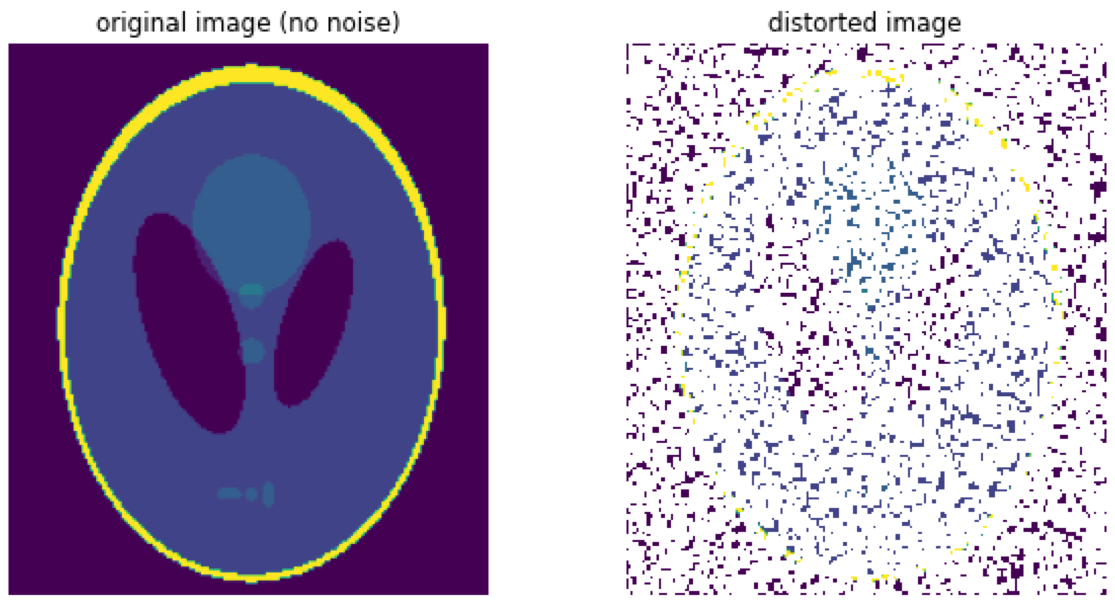
3 Can we do this on images?

[13]:
```
x = np.load('shepp_logan_phantom.npy')
x = x/x.max()
ny, nx = x.shape
```

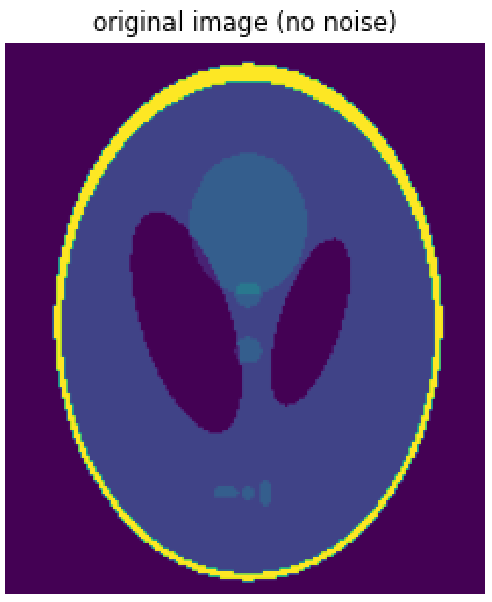
[14]:
```
Dop = \
    [pylops.FirstDerivative(ny * nx, dims=(ny,nx),dir=0, edge=False,
                            kind='backward', dtype=np.double),
     pylops.FirstDerivative(ny * nx, dims=(ny,nx),dir=1, edge=False,
                            kind='backward', dtype=np.double)]
```

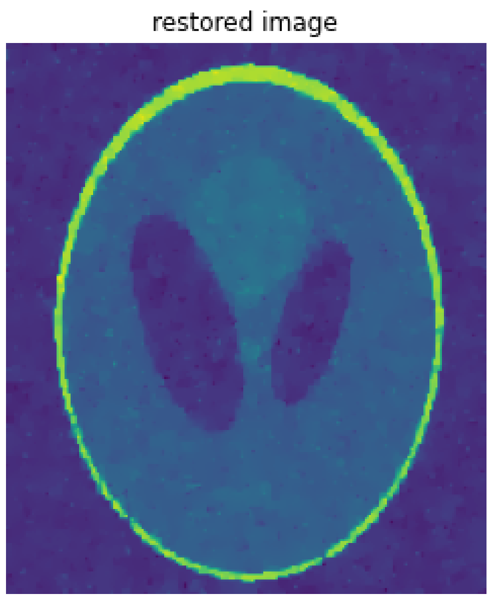
This time, we use as example image restoration. A random part of the image was lost (40%). We try to restore it.

[15]:
```
perc_subsampling = 0.6
nxsub = int(np.round(ny*nx*perc_subsampling))
iava = np.sort(np.random.permutation(np.arange(ny*nx))[:nxsub])
Rop = pylops.Restriction(ny*nx, iava, dtype=np.double)
y=Rop * x.flatten()
dev=0.1
n = np.random.normal(0, dev, y.size)
y=y+n

plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.imshow(x)
#plt.legend(['$u_\alpha^+$'])
plt.title('original image (no noise)')
plt.axis('off')
plt.subplot(122)
ymask = Rop.mask((x.flatten()))
ymask = ymask.reshape(ny, nx)
plt.imshow(ymask)
plt.title('distorted image');
plt.axis('off');
```



```
[16]: mu = 2
lamda = [0.3, 0.3]
niter = 20
niterinner = 10
xinv, niter = \
    pylops.optimization.sparsity.SplitBregman(Rop, Dop, y.flatten(),
                                              niter, niterinner,
                                              mu=mu, epsRL1s=lamda,
                                              tol=1e-4, tau=1., show=False,
                                              **dict(iter_lim=5, damp=1e-4))

plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.imshow(x)
#plt.legend(['$u_\alpha^+$'])
plt.title('original image (no noise)')
plt.axis('off')
plt.subplot(122)
plt.imshow(xinv.reshape(ny,nx))
plt.title('restored image');
plt.axis('off');
```





Contains problematic spots.

```
[ ]:
```

Regularization

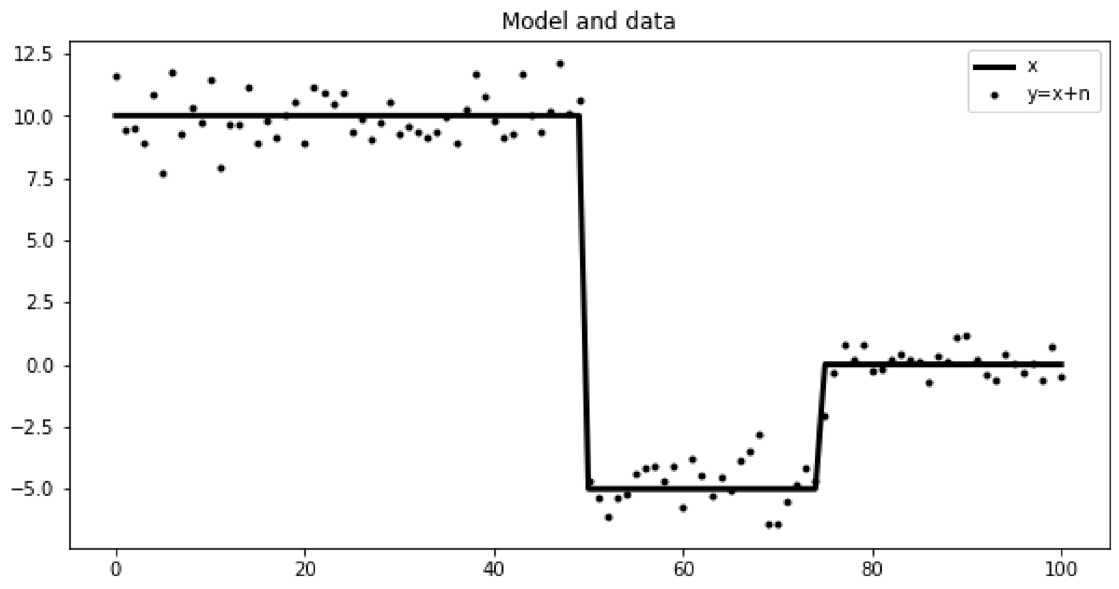
January 31, 2021

```
[1]: #!pip install --user pylops
```

```
[2]: # sphinx_gallery_thumbnail_number = 5
     import numpy as np
     import matplotlib.pyplot as plt

     import pylops

     plt.close('all')
     np.random.seed(1)
```

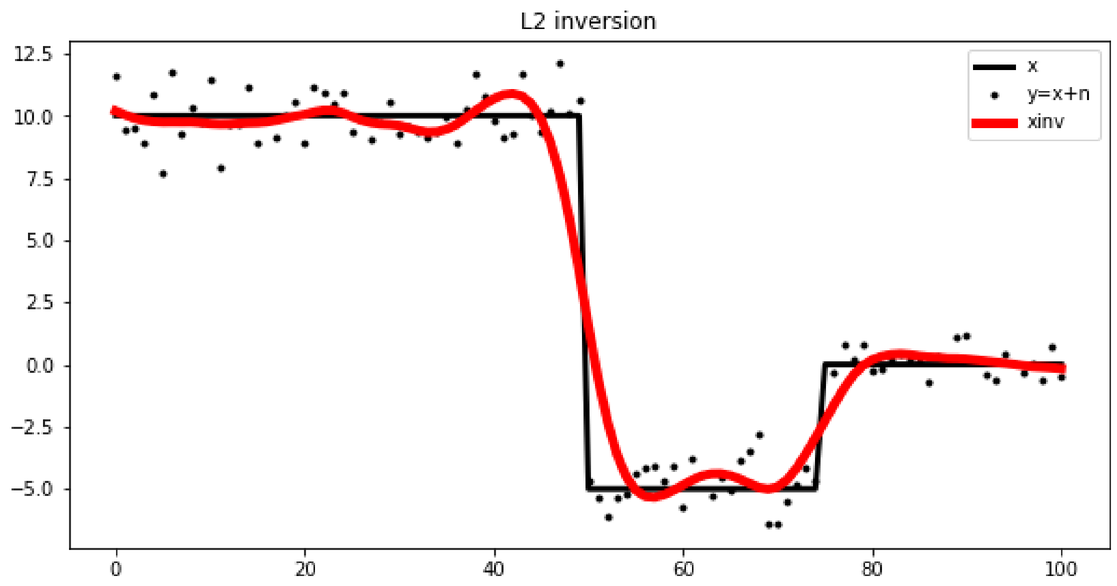
```
[3]: nx = 101
     x = np.zeros(nx)
     x[:nx//2] = 10
     x[nx//2:3*nx//4] = -5

     Iop = pylops.Identity(nx)

     n = np.random.normal(0, 1, nx)
     y = Iop*(x + n)

     plt.figure(figsize=(10, 5))
     plt.plot(x, 'k', lw=3, label='x')
     plt.plot(y, '.k', label='y=x+n')
     plt.legend()
     plt.title('Model and data')
```

```
[3]: Text(0.5, 1.0, 'Model and data')
```



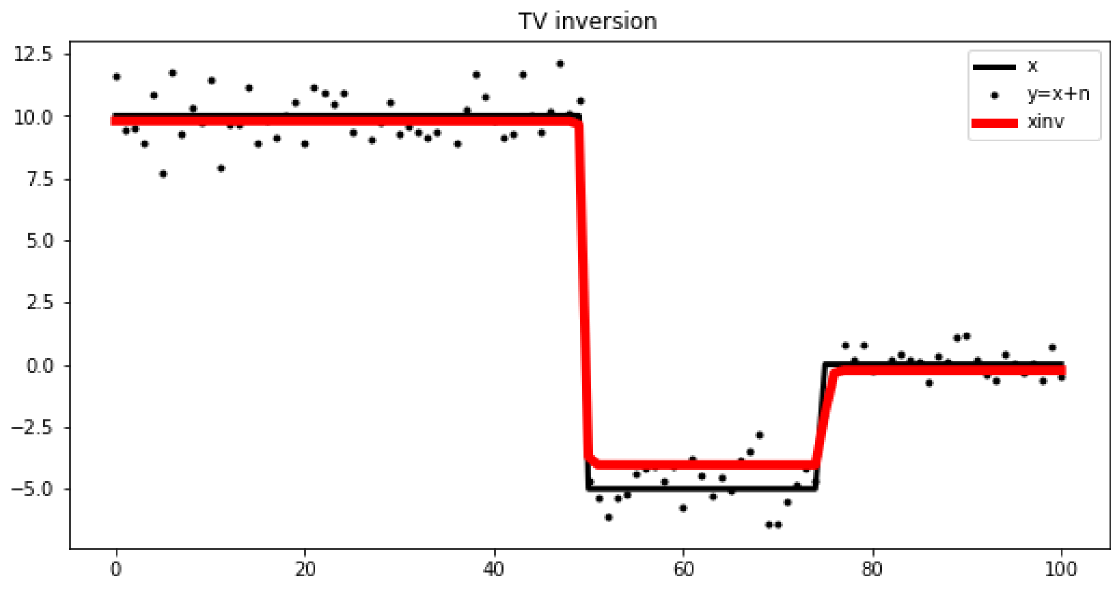
```
[4]: D2op = pylops.SecondDerivative(nx, edge=True)
     lamda = 1e2

     xinv = \
         pylops.optimization.leastsquares.RegularizedInversion(Iop, [D2op], y,
                                                               epsRs=[np.sqrt(lamda/
       ↪2)],
                                                               **dict(iter_lim=30))

     plt.figure(figsize=(10, 5))
     plt.plot(x, 'k', lw=3, label='x')
     plt.plot(y, '.k', label='y=x+n')
     plt.plot(xinv, 'r', lw=5, label='xinv')
     plt.legend()
     plt.title('L2 inversion')
```

```
[4]: Text(0.5, 1.0, 'L2 inversion')
```



[5]:
```
Dop = pylops.FirstDerivative(nx, edge=True, kind='backward')
mu = 0.01
lamda = 0.3
niter_out = 50
niter_in = 3

xinv, niter = \
    pylops.optimization.sparsity.SplitBregman(Iop, [Dop], y, niter_out,
                                              niter_in, mu=mu, epsRL1s=[lamda],
                                              tol=1e-4, tau=1.,
                                              **dict(iter_lim=30, damp=1e-10))

plt.figure(figsize=(10, 5))
plt.plot(x, 'k', lw=3, label='x')
plt.plot(y, '.k', label='y=x+n')
plt.plot(xinv, 'r', lw=5, label='xinv')
plt.legend()
plt.title('TV inversion')
```

[5]:
```
Text(0.5, 1.0, 'TV inversion')
```



[ ]:
```
x = np.load('shepp_logan_phantom.npy')
x = x/x.max()
ny, nx = x.shape

perc_subsampling = 0.6
nxsub = int(np.round(ny*nx*perc_subsampling))
iava = np.sort(np.random.permutation(np.arange(ny*nx))[:nxsub])
Rop = pylops.Restriction(ny*nx, iava, dtype=np.complex)
# Fop = pylops.signalprocessing.FFT2D(dims=(ny, nx))

n = np.random.normal(0, 0., (ny, nx))
y = Rop*(x.flatten() + n.flatten())
# yfft = Fop*(x.flatten() + n.flatten())
# yfft = np.fft.fftshift(yfft.reshape(ny, nx))
yfft=x.reshape(ny,nx)

ymask = Rop.mask((x.flatten()) + n.flatten())
ymask = ymask.reshape(ny, nx)
ymask.data[:] = np.fft.fftshift(ymask.data)
ymask.mask[:] = np.fft.fftshift(ymask.mask)

fig, axs = plt.subplots(1, 3, figsize=(14, 5))
axs[0].imshow(x, vmin=0, vmax=1, cmap='gray')
axs[0].set_title('Model')
axs[0].axis('tight')
axs[1].imshow(np.abs(yfft), vmin=0, vmax=1, cmap='rainbow')
axs[1].set_title('Full data')
```

```
axs[1].axis('tight')
axs[2].imshow(np.abs(ymask), vmin=0, vmax=1, cmap='rainbow')
axs[2].set_title('Sampled data')
axs[2].axis('tight')
```

[26]:
```
Iop = pylops.Identity(ny*nx)
Dop = \
    [pylops.FirstDerivative(ny * nx, dims=(ny, nx), dir=0, edge=False,
                            kind='backward', dtype=np.complex),
     pylops.FirstDerivative(ny * nx, dims=(ny, nx), dir=1, edge=False,
                            kind='backward', dtype=np.complex)]

# TV
mu = 0.3
lamda = [0.1, 0.1]
niter = 20
niterinner = 10

xinv, niter = \
    pylops.optimization.sparsity.SplitBregman(Iop, Dop, yfft.flatten(),
                                              niter, niterinner,
                                              mu=mu, epsRL1s=lamda,
                                              tol=1e-4, tau=1., show=False,
                                              **dict(iter_lim=5, damp=1e-4))
xinv = np.real(xinv.reshape(ny, nx))

fig, axs = plt.subplots(1, 2, figsize=(9, 5))
axs[0].imshow(x, vmin=0, vmax=1, cmap='gray')
axs[0].set_title('Model')
axs[0].axis('tight')
axs[1].imshow(xinv, vmin=0, vmax=1, cmap='gray')
axs[1].set_title('TV Inversion')
axs[1].axis('tight')

fig, axs = plt.subplots(2, 1, figsize=(10, 5))
axs[0].plot(x[ny//2], 'k', lw=5, label='x')
axs[0].plot(xinv[ny//2], 'r', lw=3, label='xinv TV')
axs[0].set_title('Horizontal section')
axs[0].legend()
axs[1].plot(x[:, nx//2], 'k', lw=5, label='x')
axs[1].plot(xinv[:, nx//2], 'r', lw=3, label='xinv TV')
axs[1].set_title('Vertical section')
axs[1].legend()
```

```
/home/wuebbel/.local/lib/python3.7/site-
packages/pylops/basicoperators/VStack.py:102: ComplexWarning: Casting complex
values to real discards the imaginary part
```

```
y[self.nnops[iop]:self.nnops[iop + 1]] = oper.matvec(x).squeeze()
```

```
---------------------------------------------------------------------------
UFuncTypeError                            Traceback (most recent call last)
<ipython-input-26-1bebec948e9d> in <module>
     17                                                 mu=mu, epsRL1s=lamda,
     18                                                 tol=1e-4, tau=1., show=False,
---> 19                                                 **dict(iter_lim=5,␣
 ↪damp=1e-4))
     20 xinv = np.real(xinv.reshape(ny, nx))
     21

~/.local/lib/python3.7/site-packages/pylops/optimization/sparsity.py in␣
 ↪SplitBregman(Op, RegsL1, data, niter_outer, niter_inner, RegsL2, dataregsL2,␣
 ↪mu, epsRL1s, epsRL2s, tol, tau, x0, restart, show, **kwargs_lsqr)
   1361                                          epsRs=epsRs,
   1362                                          x0=x0 if restart else xinv,
-> 1363                                          **kwargs_lsqr)
   1364             # Shrinkage
   1365             d = [_softthreshold(RegsL1[ireg] * xinv + b[ireg],␣
 ↪epsRL1s[ireg])

~/.local/lib/python3.7/site-packages/pylops/optimization/leastsquares.py in␣
 ↪RegularizedInversion(Op, Regs, data, Weight, dataregs, epsRs, x0, returninfo,␣
 ↪**kwargs_solver)
    325     if ncp == np:
    326         xinv, istop, itn, r1norm, r2norm = lsqr(RegOp, datatot,
--> 327                                                 **kwargs_solver)[0:5]
    328     else:
    329         xinv, istop, itn, r1norm, r2norm = \

/opt/conda/lib/python3.7/site-packages/scipy/sparse/linalg/isolve/lsqr.py in␣
 ↪lsqr(A, b, damp, atol, btol, conlim, iter_lim, show, calc_var, x0)
    372     if beta > 0:
    373         u = (1/beta) * u
--> 374         v = A.rmatvec(u)
    375         alfa = np.linalg.norm(v)
    376     else:

~/.local/lib/python3.7/site-packages/pylops/LinearOperator.py in rmatvec(self, x)
    160             raise ValueError('dimension mismatch')
    161
--> 162         y = self._rmatvec(x)
    163
    164         if x.ndim == 1:
```

```
~/.local/lib/python3.7/site-packages/pylops/basicoperators/VStack.py in␣
 ↪_rmatvec(self, x)
    107         y = ncp.zeros(self.mops, dtype=self.dtype)
    108         for iop, oper in enumerate(self.ops):
--> 109             y += oper.rmatvec(x[self.nnops[iop]:self.nnops[iop + 1]]).
 ↪squeeze()
    110         return y

UFuncTypeError: Cannot cast ufunc 'add' output from dtype('complex128') to␣
 ↪dtype('float64') with casting rule 'same_kind'
```

[16]:
```
ny*nx*3
```

[16]:
```
99450
```

[24]:
```
y.shape
```

[24]:
```
(19890,)
```

[25]:
```
nx*ny
```

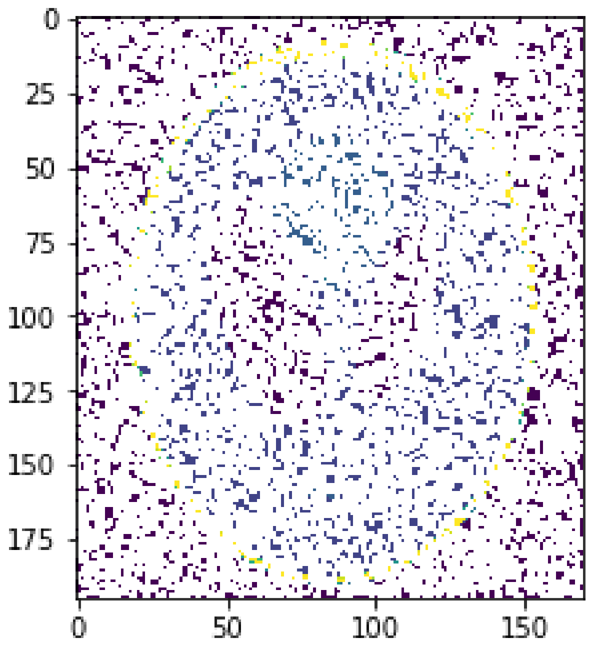
[25]:
```
33150
```

[102]:
```
x = np.load('shepp_logan_phantom.npy')
x = x/x.max()
ny, nx = x.shape

Dop = \
    [pylops.FirstDerivative(ny * nx, dims=(ny,nx),dir=0, edge=False,
                            kind='backward', dtype=np.double),
     pylops.FirstDerivative(ny * nx, dims=(ny,nx),dir=1, edge=False,
                            kind='backward', dtype=np.double)]

perc_subsampling = 0.8
nxsub = int(np.round(ny*nx*perc_subsampling))
iava = np.sort(np.random.permutation(np.arange(ny*nx))[:nxsub])
Rop = pylops.Restriction(ny*nx, iava, dtype=np.double)
y=Rop * x.flatten()

mu = 2
lamda = [0.3, 0.3]
niter = 20
niterinner = 10

xinv, niter = \
    pylops.optimization.sparsity.SplitBregman(Rop, Dop, y.flatten(),
                                              niter, niterinner,
                                                   mu=mu, epsRL1s=lamda,
                                                   tol=1e-4, tau=1., show=False,
                                                   **dict(iter_lim=5, damp=1e-4))
```

[103]:
```
perc_subsampling = 0.6
nxsub = int(np.round(ny*nx*perc_subsampling))
iava = np.sort(np.random.permutation(np.arange(ny*nx))[:nxsub])
Rop = pylops.Restriction(ny*nx, iava, dtype=np.double)
y=Rop * x.flatten()
print(y.size,x.size)
ymask = Rop.mask((x.flatten()))
ymask = ymask.reshape(ny, nx)
plt.imshow(ymask)
```

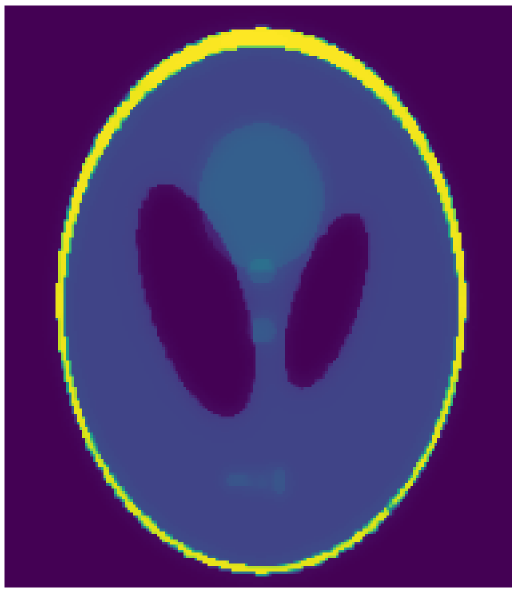
```
19890 33150
```

[103]:
```
<matplotlib.image.AxesImage at 0x7f887ac98250>
```



[104]:
```
plt.imshow(xinv.reshape(ny,nx))
plt.axis('off')
np.max(xinv)
```

[104]:
```
0.9839150825923475
```



```
[105]: iava[1:10]
```

```
[105]: array([ 2,  3,  5,  6,  7,  8,  9, 10, 11])
```

```
[ ]:
```

Bessel

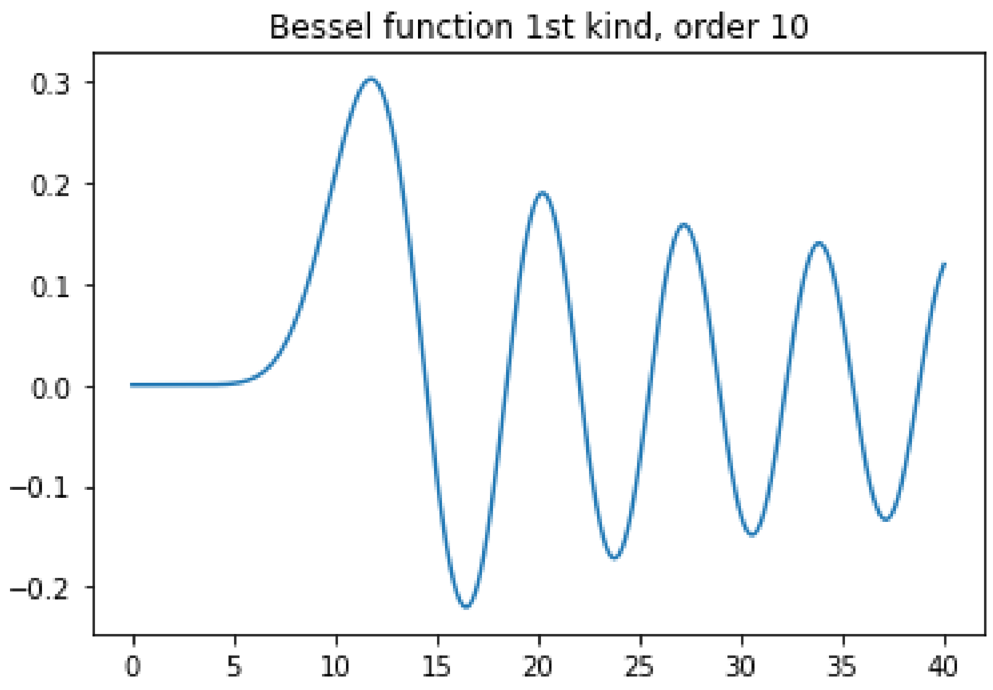
January 31, 2021

1 Bessel Function J_n of integer order

[9]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy
from scipy import special
```

[19]:
```
n=0
N=1024
order=10
X=np.linspace(0,40,N)
Y=special.jn(order,X)
plt.plot(X,Y)
plt.title('Bessel function 1st kind, order '+str(order))
```

[19]:
```
Text(0.5, 1.0, 'Bessel function 1st kind, order 10')
```



```
[ ]:
```

radon

January 31, 2021

1 Radon Transform: Basic utilities

This file provides some basic utilities for 2D Radon Transform. At least the analytical ones were not easily available.

##Exported functions:

To import functions, set radon_interactive = False and include: %run radon.ipynb

plot functions:

- init_canvas(N,M,a,b,c,d) - initialize global canvas of size $N \times M$ on $[a, b] \times [c, d]$ in R^2
- clear_canvas()
- show_canvas()
- draw_pixel(x,y)
- draw_line(phi,s)
- draw_scene(scene) - draws ellipses, rectangles, exps. For definition, see below.
- parproj(scene,p,q,phivec) - perform Radon Transform on scene. p or phivec must be given. Use named parameters.
- parproj_canvas(p,q,phivec,oversample) - perform Radon Transform on current canvas. Use oversample if q is small.

```
[1]: import numpy as np
     import matplotlib.pyplot as plt
     import math
     import scipy.interpolate

     import warnings
     warnings.simplefilter("error")
     warnings.filterwarnings("ignore", category=DeprecationWarning)

     try: radon_interactive
     except NameError: radon_interactive = True
```

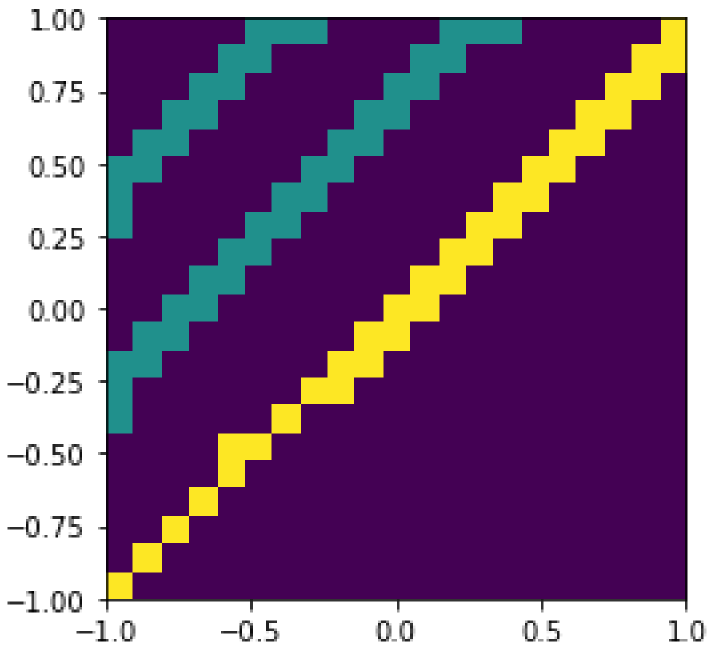
We start with some simple drawing utilities. Our canvas has $(2N + 1) \times (2M + 1)$ points and covers $[a, b] \times [c, d]$. X, Y have the x and y components of each point.

```
[2]: def init_canvas(N,M=None,a=-1,b=1,c=-1,d=1):
         global␣
      ↪canvas,canvas_X,canvas_Y,canvas_a,canvas_b,canvas_c,canvas_d,canvas_N,canvas_M
```

```
    if (M is None):
        M=N

    canvas=np.zeros([2*M+1,2*N+1])
    canvas_a=a
    canvas_b=b
    canvas_c=c
    canvas_d=d
    canvas_N=N
    canvas_M=M

    #X=np.linspace(a,b,2*N+1,endpoint=True).reshape(2*N+1,1)
    #Y=np.linspace(d,c,2*M+1,endpoint=True).reshape(2*M+1,1)
    h=(b-a)/(2*N+1)
    X=np.arange(0,2*N+1)*h+a+h/2
    h=(c-d)/(2*M+1)
    Y=np.arange(0,2*M+1)*h+d+h/2
    canvas_X,canvas_Y=np.meshgrid(X,Y)

def clear_canvas():
    global canvas
    canvas=canvas*0

def show_canvas():
    global canvas
    extent=canvas_a,canvas_b,canvas_c,canvas_d
    return plt.imshow(canvas,extent=extent)

def canvas_update(I,value=1,assign=False):
    global canvas
    if (assign):
        canvas[I]=value
    else:
        canvas[I]+=value

# Computes indices into canvas. Attention: (y,x)!
def pixel_dot(x,y):
    y=(canvas_d-y)/(canvas_d-canvas_c)*(2*canvas_M)+0.5
    y=int(y)
    x=(x-canvas_a)/(canvas_b-canvas_a)*(2*canvas_N)+0.5
    x=int(x)
    return x,y

def draw_pixel(x,y,value=1):
    global canvas
    x,y=pixel_dot(x,y)
    if (x>=0) and (y>=0) and (x<=2*canvas_N) and (y<=2*canvas_M):
        canvas[y,x]=value

# We should have a reasonable way of drawing a line. Later.
# horrible implementation.
def theta(phi):
    return np.array([math.cos(phi),math.sin(phi)])

def thetaperp(phi):
    return theta(phi+math.pi/2)

def draw_line(phi,s,value=1,a=np.zeros(2)):
    maxlen=max(abs(canvas_a),abs(canvas_b),abs(canvas_c),abs(canvas_d))*math.
 ↪sqrt(2)
    maxind=max(canvas_N,canvas_M)*4
    h=maxlen/maxind
    s1=(s+a.dot(theta(phi)))
    t1=s1*theta(phi)
    t2=thetaperp(phi)
    for i in range(-maxind,maxind):
        x=t1+i*h*t2
        draw_pixel(x[0],x[1],value=value)
```

[3]:
```
if radon_interactive:
    init_canvas(10,10,-1,1,-1,1)
    draw_line(math.pi/4*3,0.5)
    draw_line(math.pi/4*3,0,value=2)
    draw_line(math.pi/4*3,1)
    show_canvas()

    init_canvas(1,2)
    print(canvas_Y)
    print(np.arange(-1,2)*1/3)
```

```
[[ 8.00000000e-01  8.00000000e-01  8.00000000e-01]
 [ 4.00000000e-01  4.00000000e-01  4.00000000e-01]
 [-5.55111512e-17 -5.55111512e-17 -5.55111512e-17]
 [-4.00000000e-01 -4.00000000e-01 -4.00000000e-01]
 [-8.00000000e-01 -8.00000000e-01 -8.00000000e-01]]
[-0.33333333  0.          0.33333333]
```



The following considerations will make life **a lot** easier:

Let $f_a(x) := f(x-a)$. Then

$$Rf_a(\theta, s) = \int_{x\theta=s} f(x-a)\,dx = \int_{x\theta=s+a\theta} f(x)\,dx = Rf(\theta, s-a\theta).$$

Let $\theta(\varphi) = (\cos\varphi, \sin\varphi)$. Let U_ψ a rotation of ψ around the origin. Let $f_\psi(x) = f(U_\psi x)$. Then

$$Rf_\psi(\theta(\varphi), s) = \int_{x\theta(\varphi)=s} f(U_\psi x)\,dx = \int_{x\theta(\varphi-\psi)=s} f(x)\,dx = Rf(\theta(\varphi-\psi), s)$$

Let A any matrix, $f_A(x) := f(Ax)$. We compute the Radon Transform of

$$Rf_A(\theta, s) = \int_{x\theta=s} f(Ax)\,dx.$$

One is tempted to simply substitute $y = Ax$. However, this is not simple because dx is the surface measure, not the measure on R^n! Thus, the integration constant is *not* simply the absolute value of the determinant (one easily checks that in R^2, scaling with a would scale the results by a^2, while in fact it only scales with a).

Instead, we first prove: Let $\delta_B(x) := \delta(Bx)$ in the usual sense: Let

$$\delta = \lim \delta_k,\ \delta_B = \lim \delta_{k,B},\ \delta_{k,B}(x) = \delta_k(Bx).$$

Then $\delta_B = \frac{1}{|\det B|}\delta$.

Proof:

$$\delta_B(\varphi) = \int_{R^n} \delta(Bx)\varphi(x)\,dx =_{x=B^{-1}y,\, dx=dy/|\det B|} = \frac{1}{|\det B|}\int_{R^n} \delta(x)\varphi(B^{-1}y)\,dy = \frac{1}{|\det B|}\delta(\varphi).$$

with $l = ||D^{-t}\theta||$, $s' = s/l$, $\theta' = (D^{-t}\theta)/l$.

Now we can easily substitute $x = A^{-1}y$. Note that δ is a function on R!

$$\begin{aligned}
Rf_A(\theta,s) &= \int_{R^n} \delta\left((x\,,\theta) - s\right) f(Ax)\,dx & x := A^{-1}y,\ dx = \frac{dy}{|\det A|}\\
&= \frac{1}{|\det A|}\int_{R^n} \delta\left((A^{-1}y,\theta) - s\right) f(y)\,dy \\
&= \frac{1}{|\det A|}\int_{R^n} \delta\left((y, A^{-t}\theta) - s\right) f(y)\,dy & L := ||A^{-t}\theta||\\
&= \frac{1}{|\det A|}\int_{R^n} \delta\left(L\left((y, \frac{A^{-t}\theta}{L}) - \frac{s}{L}\right)\right) f(y)\,dy \\
&= \frac{1}{|\det A|}\frac{1}{L}\int_{R^n} \delta\left((y, \frac{A^{-t}\theta}{L}) - \frac{s}{L}\right) f(y)\,dy \\
&= \frac{1}{|\det A|}\frac{1}{||A^{-t}\theta||}(Rf)(\frac{A^{-t}\theta}{||A^{-t}\theta||}, \frac{s}{||A^{-t}\theta||})
\end{aligned}$$

In particular: Set $y = \lambda x$, then $dx = \frac{1}{\lambda^{n-1}}\,dy$.

We start with the Radon transform of $f(x) := \lambda e^{-||\lambda x - a||/2}$. For $a = 0$, we have

$$\begin{aligned}
\int_{R^2} f(x)\,dx &= \int_{R^2} \lambda e^{-||\lambda x||^2/2}\,dx & y = \lambda x,\ x = \frac{y}{\lambda},\ dx = \frac{1}{\lambda^2}\,dy\\
&= \frac{1}{\lambda}\int_{R^2} e^{-||y||^2/2}\,dy \\
&= \frac{2\pi}{\lambda}
\end{aligned}$$

For $\lambda = 1$, $a = 0$:

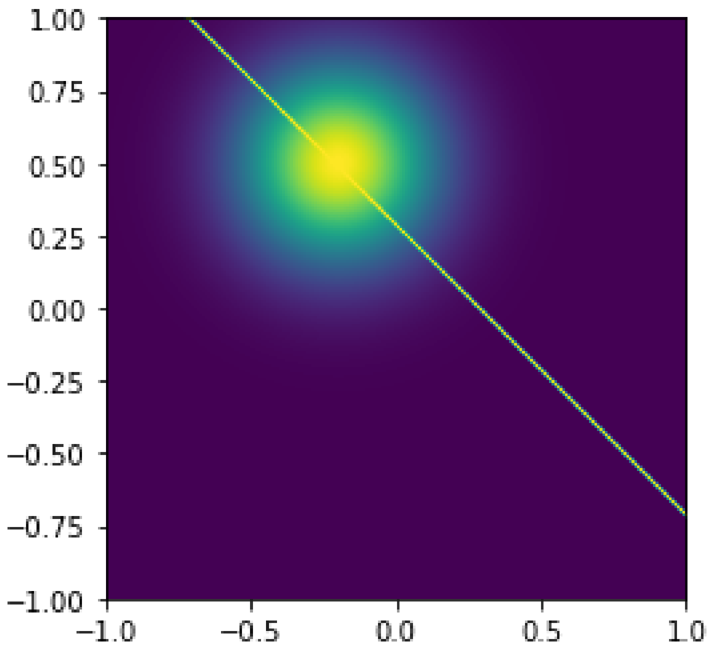
$$\begin{aligned}
Rf(\theta,s) &= \int_{x\theta=s} e^{-||x||^2/2}dx \\
&= \int_R e^{-||s\theta + t\theta^\perp||^2/2}dt \\
&= e^{-s^2/2}\int_R e^{-t^2/2}dt \\
&= e^{-s^2/2}\sqrt{2\pi}
\end{aligned}$$

Now use the formula for translated and scaled Radon Transform from above.

[4]:
```
def radon_exp(phi,s,a=np.array([0,0]),lamda=1):
    s=(s-a.dot(theta(phi)))
    return math.sqrt(2*math.pi)*math.exp(-lamda*lamda*s*s)

def draw_exp(a=np.array([0,0]),lamda=1,value=1):
    global canvas
    R=lamda*np.hypot(canvas_Y-a[1],canvas_X-a[0])
    canvas=canvas+lamda*np.exp(-R*R)*value
```

[5]:
```
if radon_interactive:
    init_canvas(128,128,-1,1,-1,1)
    #clear_canvas()
    lamda=3
    phi=math.pi/4
    s=0.2
    a=np.array([-0.2,0.5])
    print(radon_exp(phi,s,a=a,lamda=lamda))
    draw_exp(a=a,lamda=lamda)
    L=np.max(canvas)
    draw_line(phi,s,value=L)
    show_canvas()
    #show_canvas()
```

```
2.503310001820181
```

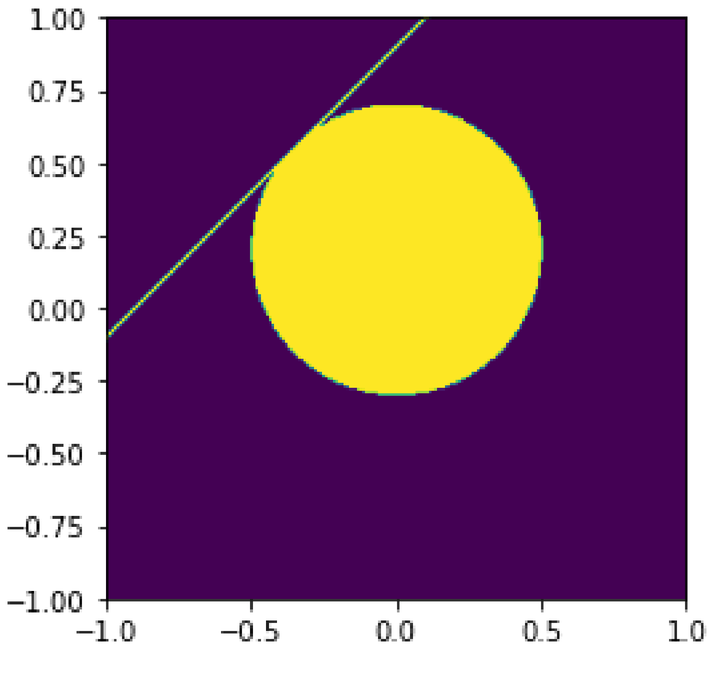


Characteristic function of a ball of radius r located at a.

[6]:
```
def radon_ball(phi,s,r=1,a=np.array([0,0])):
    s=s-a.dot(theta(phi))
    D=r*r-s*s
    if (D<0):
        return 0
    else:
        return 2*math.sqrt(D)

def draw_ball(r=1,a=np.array([0,0]),value=1,assign=False):
    ax=a[0]
    ay=a[1]
    I=np.where((canvas_X-ax)*(canvas_X-ax)+(canvas_Y-ay)*(canvas_Y-ay)<=r*r)
    canvas_update(I,value,assign)
```

[7]:
```
if radon_interactive:
    init_canvas(128,128,-1,1,-1,1)
    r=0.5
    a=np.array([0,0.2])
    draw_ball(r,a)
    phi=math.pi/4*3
    s=0.640
    print(radon_ball(phi,s,r=r,a=a))
    draw_line(phi,s)
    show_canvas()
```

```
0.07534815150370204
```



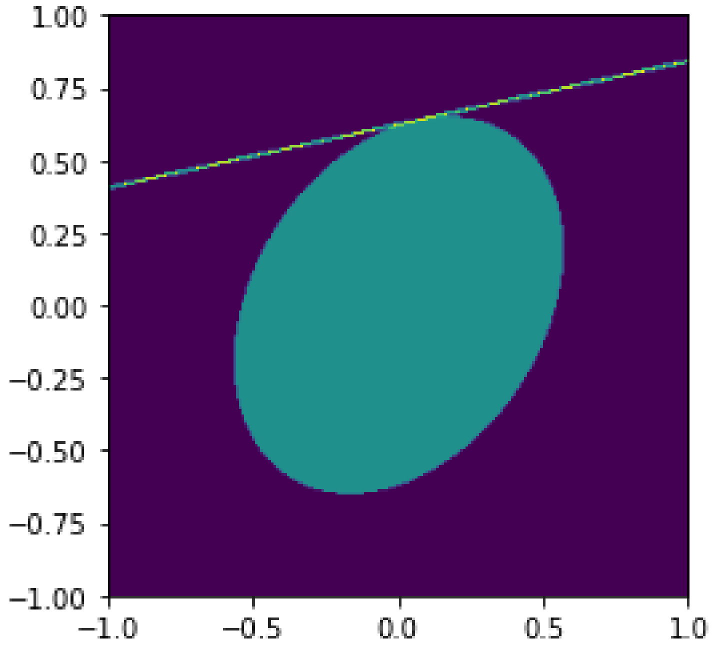
Characteristic function of an ellipse with half axes r_x and r_y, located at a, rotated ψ degrees. This is not correct yet, but shows the idea. Forgetting about the rotation and the translation, we have $f_{\text{Ellipse}}(x) = f_D(x) = f(Dx)$ with $D = diag(1/r_x, 1/r_y)$. According to the lemma above, the integration factor is

$$\frac{r_x\, r_y}{|| \begin{pmatrix} \cos\theta\, r_x \\ \sin\theta\, r_y \end{pmatrix} ||}.$$

[8]:
```
def radon_ellipse(phi,s,rx=1,ry=1,psi=0,a=np.array([0,0])):
    s=s-a.dot(theta(phi))
    phi=phi-psi

    x=math.cos(phi)*rx
    y=math.sin(phi)*ry
    l=math.hypot(x,y)
    phi=math.atan2(y,x)

    # We have D=(1/rx,1/ry)
    Det=(rx*ry)/l

    s=s/l
    return Det*radon_ball(phi,s,1)

def draw_ellipse(rx=1,ry=1,psi=0,a=np.zeros(2),value=1,assign=False):
    X=math.cos(psi)*(canvas_X-a[0])+math.sin(psi)*(canvas_Y-a[1]);
    Y=-math.sin(psi)*(canvas_X-a[0])+math.cos(psi)*(canvas_Y-a[1]);
    I=np.where(X*X/(rx*rx)+Y*Y/(ry*ry)<=1)
    canvas_update(I,value,assign)
```

[9]:
```
if radon_interactive:
    init_canvas(128,128,-1,1,-1,1)
    rx=0.7
    ry=0.5
    a=np.array([0.0,0.0])
    psi=math.pi/4*0+1
    draw_ellipse(rx,ry,psi=psi,a=a)

    phi=math.pi/4+1
    s=0.61
    draw_line(phi,s,value=2)
    print(radon_ellipse(phi,s,rx,ry,psi=psi,a=a))

    show_canvas()
```

```
0.0
```



Characteristic function of a square located at 0. Undefined if θ is axis parallel and $|s| = 1$.

We do this maximally slow at this point.

Let $\theta = \theta(\varphi) = \begin{pmatrix} \cos\varphi \\ \sin\varphi \end{pmatrix}$, $\theta^\perp = \theta(\varphi + \pi/2) = \begin{pmatrix} -\sin\varphi \\ \cos\varphi \end{pmatrix}$. We compute

$$Rf(\theta(\varphi), s) = \int_R f(s\,\theta + t\,\theta^\perp)dt$$

We rotate such that θ is in the first quadrant. Makes no difference since the unit square is rotationally invariant for right angles. Also, if θ is axis parallel, we slightly rotate it. This is dangerous and can easily be avoided, but we don't care at this point.

We check when this line crosses $x = 1$.

$$s\ \cos(\varphi) - l_x \sin\varphi = 1 \Rightarrow l_x = \frac{s\cos\varphi - 1}{\sin\varphi}.$$

We check when this line crosses $y = 1$.

$$s\ \sin(\varphi) + u_y \cos\varphi = 1 \Rightarrow u_y = \frac{1 - s\sin\varphi}{\cos\varphi}.$$

Same for -1. Now the line that crosses the unit square is for t in $[\max(l_x, l_y), \min(u_x, u_y)]$. We return the absolute value of the difference.

[10]:
```python
def radon_unit_square(phi,s):
    if (s*s>2):
        return 0
    # We make sure that phi is in the first quadrant. Makes no difference since␣
 ↪the
    # unit square is rotationally invariant for right angles.
    # Then both cosine and sine are positive.

    phi=phi+2*math.pi
    phi=phi-(phi//(math.pi/2))*math.pi/2

    cos=math.cos(phi)
    sin=math.sin(phi)
    if (cos==0):
        cos=1e-8
    if (sin==0):
        sin=1e-8
    lx=(s*cos-1)/sin
    ux=(s*cos+1)/sin
    ly=(-1-s*sin)/cos
    uy=(1-s*sin)/cos
    len=min(ux,uy)-max(lx,ly)
    if (len>0):
        return len
    else:
        return 0
```

[11]:
```python
if radon_interactive:
    print(radon_unit_square(math.pi/4,0))
```

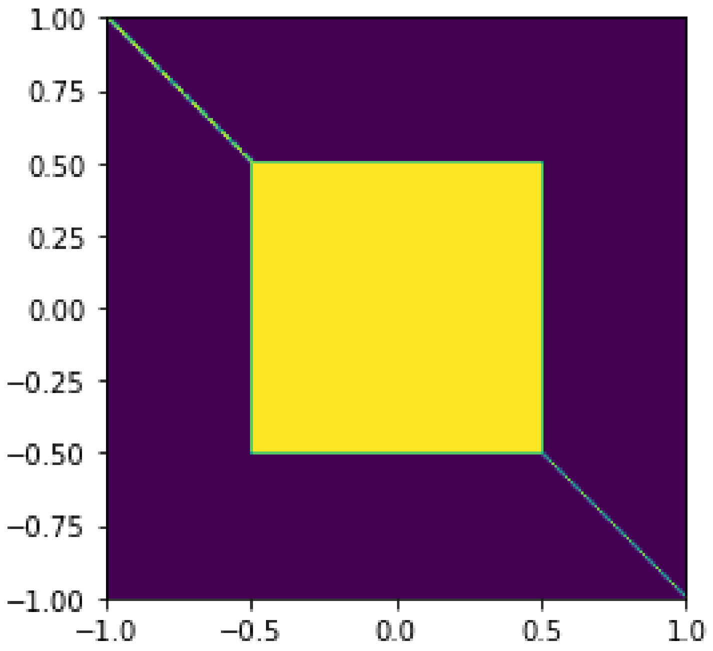
```
2.82842712474619
```

Same for pixel with its center located at a, length r_1, r_2.

[12]:
```python
def radon_rect(phi,s,rx,ry,a=np.array([0,0]),psi=0):
    s=s-a.dot(theta(phi))
    phi=phi-psi

    x=math.cos(phi)*rx
    y=math.sin(phi)*ry
    l=math.hypot(x,y)
    phi=math.atan2(y,x)

    # We have D=(1/rx,1/ry)
    Det=(rx*ry)/l

    s=s/l
    val=Det*radon_unit_square(phi,s)
    return val


def draw_rect(r1,r2,a=np.zeros(2),psi=0,value=1,assign=False):
    global canvas
    X=np.abs((math.cos(psi)*(canvas_X-a[0])+math.sin(psi)*(canvas_Y-a[1]))/r1)
    Y=np.abs((-math.sin(psi)*(canvas_X-a[0])+math.cos(psi)*(canvas_Y-a[1]))/r2)
    I=np.where((np.abs(Y)<1) & (np.abs(X)<1))
    canvas_update(I,value,assign)
```

[13]:
```
if radon_interactive:
    psi=math.pi/8*0
    a=np.array([0,0])
    init_canvas(128,128)
    clear_canvas()
    r1=1/2
    r2=1/2
    phi=math.pi/8*2
    psi=math.pi/8*0
    s=0.0
    draw_rect(r1,r2,psi=psi,a=a)
    draw_line(phi,s)
    show_canvas()

    print(radon_rect(phi,s,r1,r2,psi=psi,a=a))
```

```
1.414213562373095
```



OK, now we have everything.

We finalize all by defining scenes:

Scenes consist of

- ellipses ('E',rx,ry,ax,ay,psi,value) **NOTE** Following industry standard, psi is now in **degrees**
- rects ('R',same)
- exponentials ('X',ax,ay,lamda)
- sinc functions **not yet implemented**

draw_scene(scene): draws scenery on canvas.

image boundaries are fixed at $[-1, 1]$. Only $n = m$ is supported, sorry.

parproj(scene,p,q): creates parallel projection data for a scene. n: Number of projections. 1/q: detector spacing.

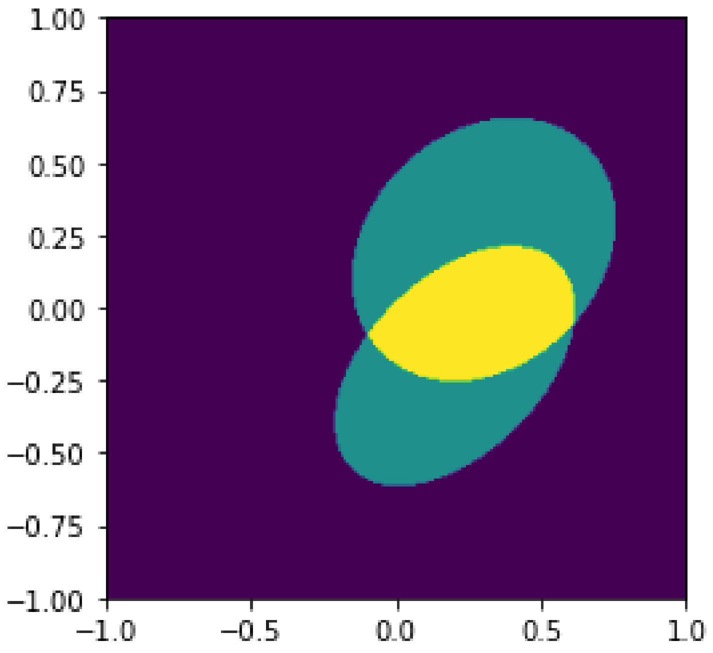
parproj_img(img,p,q): creates parallel projection data for an image. This is horribly slow, of course, and only done because we can. We will do **much** better in the coming lectures.

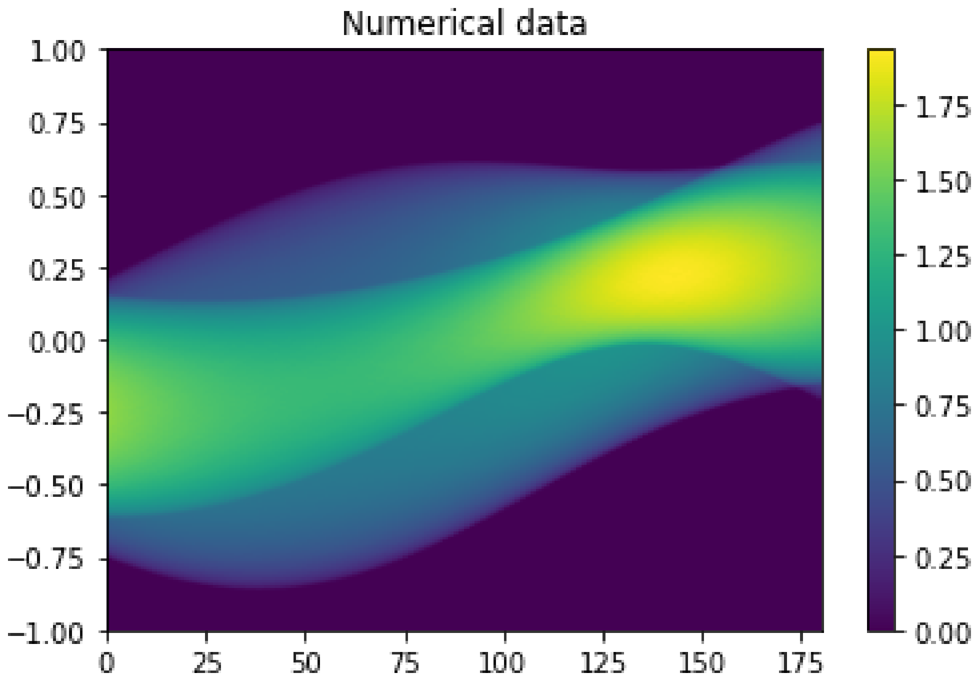
[14]:
```python
def draw_scene(scene):
    for O in scene:
        if (O[0])=='E':
            draw_ellipse(O[1],O[2],a=np.array([O[3],O[4]]),psi=O[5]*math.pi/
 ↪180,value=O[6])
            continue
        if (O[0])=='R':
            draw_rect(O[1],O[2],a=np.array([O[3],O[4]]),psi=O[5]*math.pi/
 ↪180,value=O[6])
            continue
        if (O[0])=='X':
            draw_exp(a=np.array([O[1],O[2]]),lamda=O[3])
            continue
        print('No such object')

def parproj(scene,p=None,q=None,phivec=None):
    if (phivec is None):
        phivec=np.arange(p)*math.pi/p
    p=phivec.size
    data=np.zeros([2*q+1,p])
    for O in scene:
        for i in range(-q,q+1):
            for j in range(0,p):
                #phi=j/p*math.pi
                phi=phivec[j]
                s=i/q
                if (O[0])=='E':
                    val=radon_ellipse(phi,s,rx=O[1],ry=O[2],a=np.
↪array([O[3],O[4]]),
                                   psi=O[5]*math.pi/180)
                    v=O[6]
                if (O[0])=='R':
                    val=radon_rect(phi,s,rx=O[1],ry=O[2],a=np.array([O[3],O[4]]),
                                   psi=O[5]*math.pi/180)
                    v=O[6]
                if (O[0])=='X':
                    val=radon_exp(phi,s,a=np.array([O[1],O[2]]),lamda=O[3])
                    v=O[4]
                data[i+q,j]+=v*val
    return data

# Compute Radon transform of canvas.
# works on [-1,1]^2 only.
def parproj_canvas(p=None,q=None,phivec=None,oversample=1):
    # Evaluate on grid of size[-q,q] on [-1,1]
    # We compute the integral on [-q,q]. Bad results if q is small.
    # Consider oversampling here!!!
    # (q=N*q and sample every Nth point)
    # Todo: Maybe we should always use the image size and interpolate later...
    # Matlab uses oversampling=2 (?)
    qneu=q*oversample
    h=2/(2*qneu+1)
    X=np.arange(0,2*qneu+1)*h-1+h/2
    Xqneu=X
    hq=2/(2*q+1)
    Xq=np.arange(0,2*q+1)*hq-1+hq/2

    eval_X,eval_Y=np.meshgrid(X,X)
    if (phivec is None):
        phivec=np.arange(p)*math.pi/p
    p=phivec.size
    data=np.zeros([2*q+1,p])
    for j in range(0,p):
        phi=-phivec[j]
        X=math.cos(phi)*eval_X+math.sin(phi)*eval_Y
        Y=-math.sin(phi)*eval_X+math.cos(phi)*eval_Y
        newcanvas=scipy.interpolate.interpn((canvas_X[0,:],np.flip(canvas_Y[:
↪,0])),canvas,(X,Y),bounds_error=False,fill_value=0)
        v=np.sum(newcanvas,axis=1)
        if (oversample==1):
            data[:,j]=v*h
        else:
            data[:,j]=scipy.interpolate.interpn([Xqneu],v,Xq)*h
    return data
```

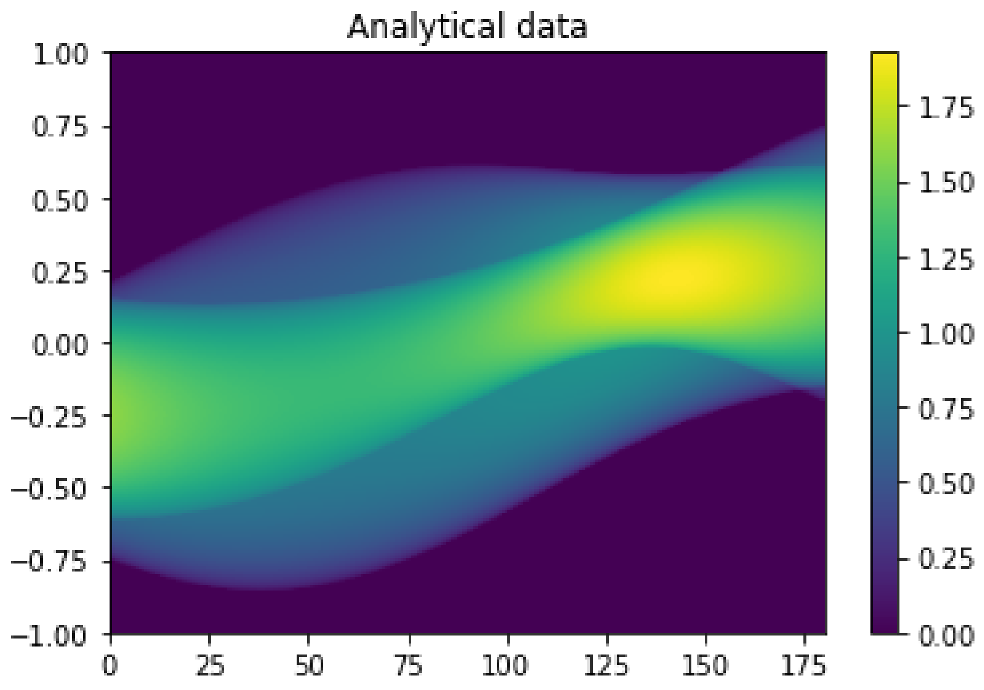
```
# More general. Correct, but do not use, too slow.
def parproj_img(image,p,q):
    data=np.zeros([2*q+1,p])
    n,m=image.shape
    hy=2/n
    hx=2/m

    z=0
    for k in range(0,n):
        for l in range(0,m):
            a=np.array([-1+hy/2+k*hy,-1+hx/2+l*hx])
            for i in range(-q,q):
                for j in range(0,p):
                    phi=j/p*math.pi
                    s=i/q
                    v=image[m-1-l,k]
                    val=radon_rect(phi,s,hx/2,hy/2,a=a)
                    data[i+q,j]+=v*val
    return data
```

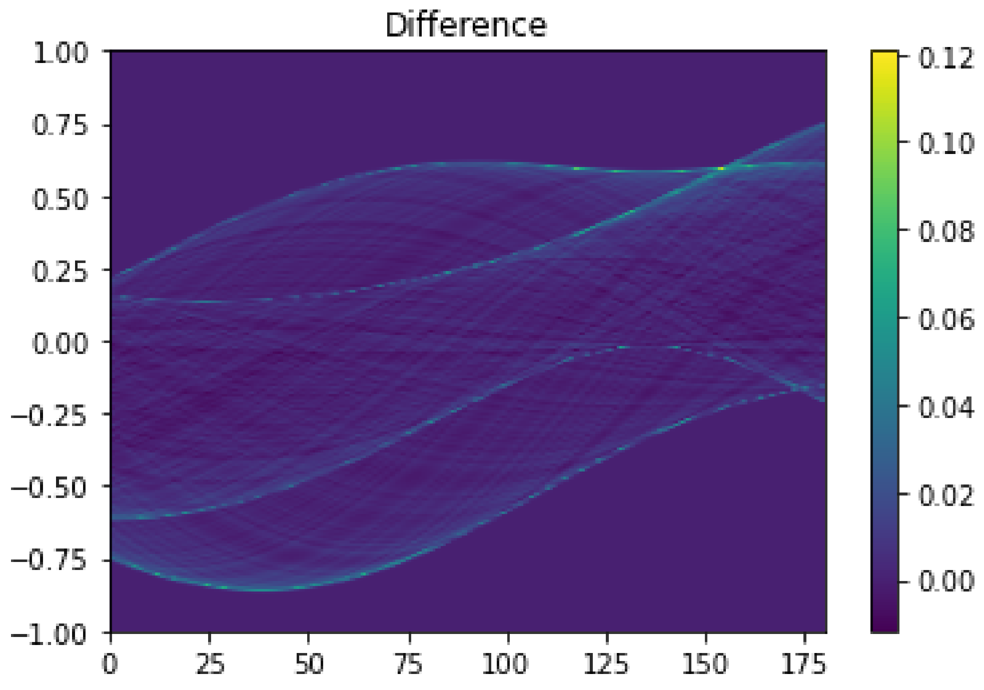
Example: Compare implementations, numerical vs. analytical.

[15]:
```
if radon_interactive:
    scene=[
            ['E',0.5,0.4,0.3,0.2,45,1],
            ['E',0.5,0.3,0.2,-0.2,45,1],

    ]
    init_canvas(128)
    draw_scene(scene)
    show_canvas()
    plt.figure()
    p=128
    q=128
    phivec=np.arange(0,p)*2*math.pi/p
    data=parproj_canvas(p=p,q=q,oversample=2);
    plt.imshow(data,extent=[0,180,-1,1],aspect='auto')
    plt.colorbar()
    plt.title('Numerical data')
    plt.figure()
    data2=parproj(scene,q=q,p=p)
    plt.imshow(data2,extent=[0,180,-1,1],aspect='auto')
    plt.colorbar()
    plt.title('Analytical data');
    plt.figure()
    plt.imshow(data-data2,extent=[0,180,-1,1],aspect='auto')
    plt.colorbar()
    plt.title('Difference')
```



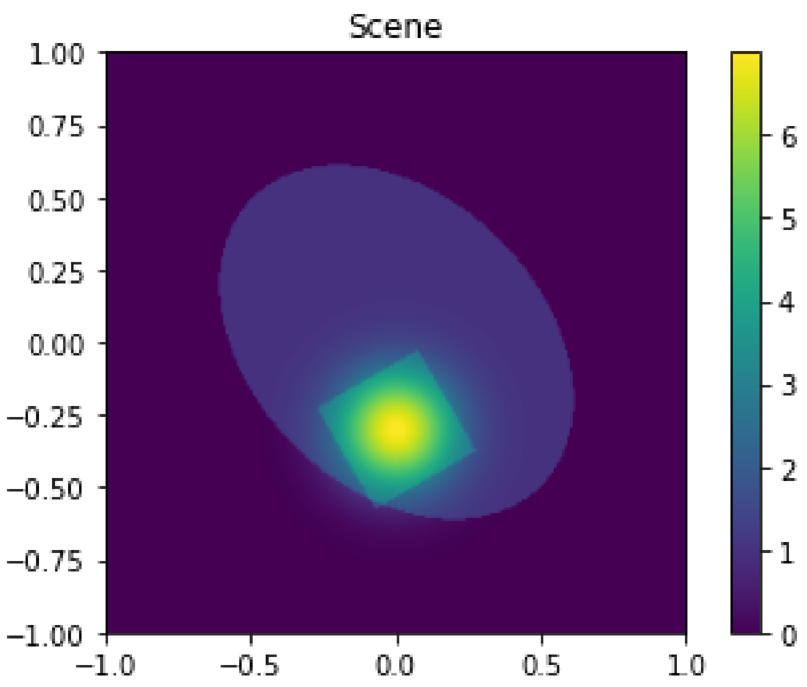


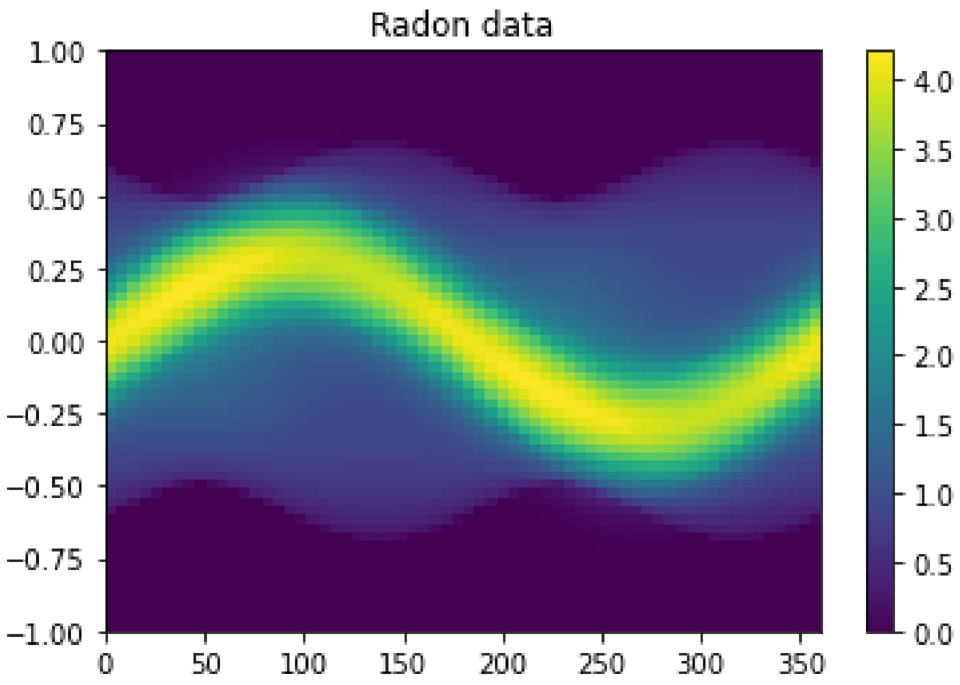


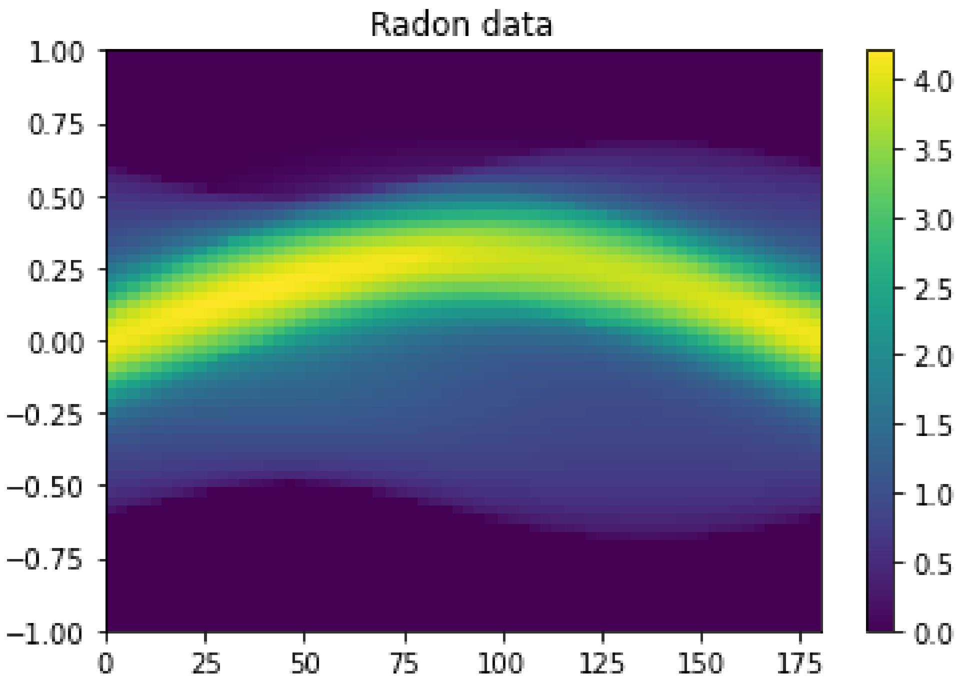


Example: Simple scene with some objects.

```
[16]: if radon_interactive:
          scene=[
              ['E',0.5,0.7,0,0,45,1],
              ['R',0.2,0.2,0,-0.3,30,1],
              ['X',-0,-0.3,5,1]
          ]

          init_canvas(128,128)
          clear_canvas()
          draw_scene(scene)
          p=64
          q=32
          phivec=np.arange(p)/p*2.*math.pi
          show_canvas()
          plt.title('Scene')
          plt.colorbar()
          plt.figure()
          extent=[0,180,-1,1]
          fullextent=[0,360,-1,1]

          data=parproj(scene,phivec=phivec,q=q)
          plt.imshow(data,extent=fullextent,aspect='auto')
          plt.colorbar()
          plt.title('Radon data')
          plt.figure()

          data=parproj(scene,p,q)
          plt.imshow(data,extent=extent,aspect='auto')
          plt.colorbar()
          plt.title('Radon data')
```


Scene
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0
0
1
2
3
4
5
6

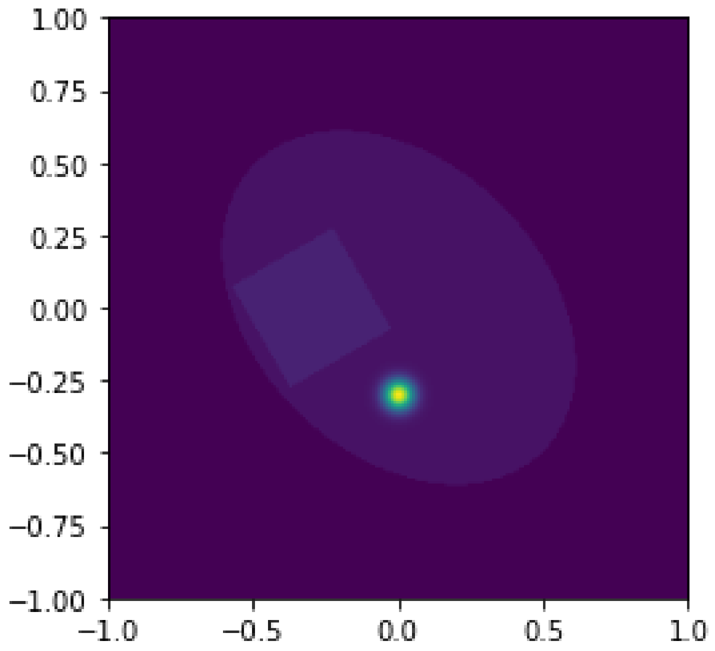

Radon data
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
50
100
150
200
250
300
350
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

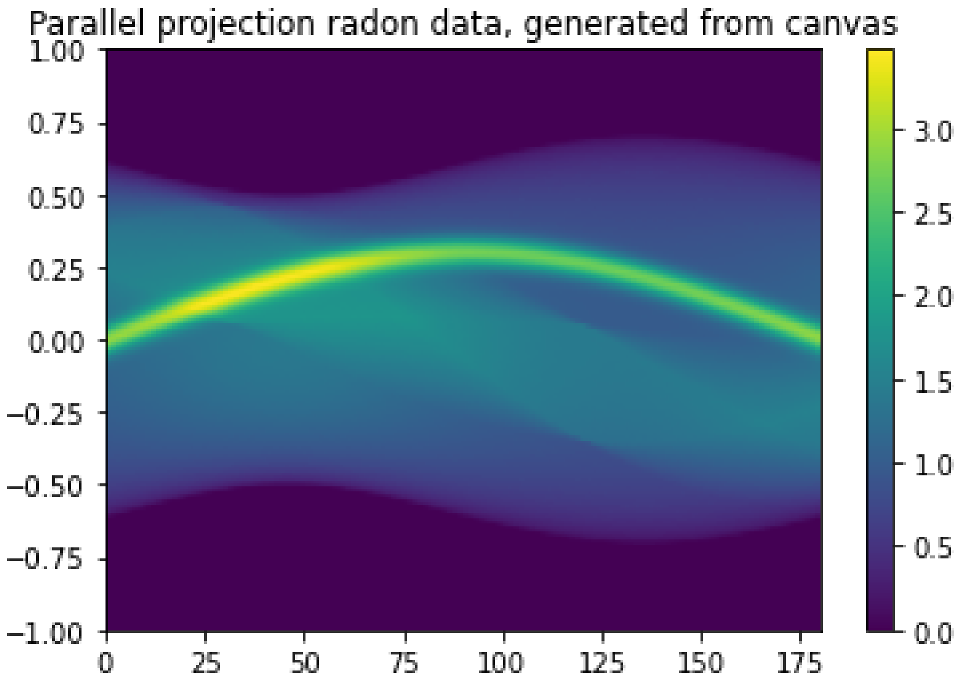


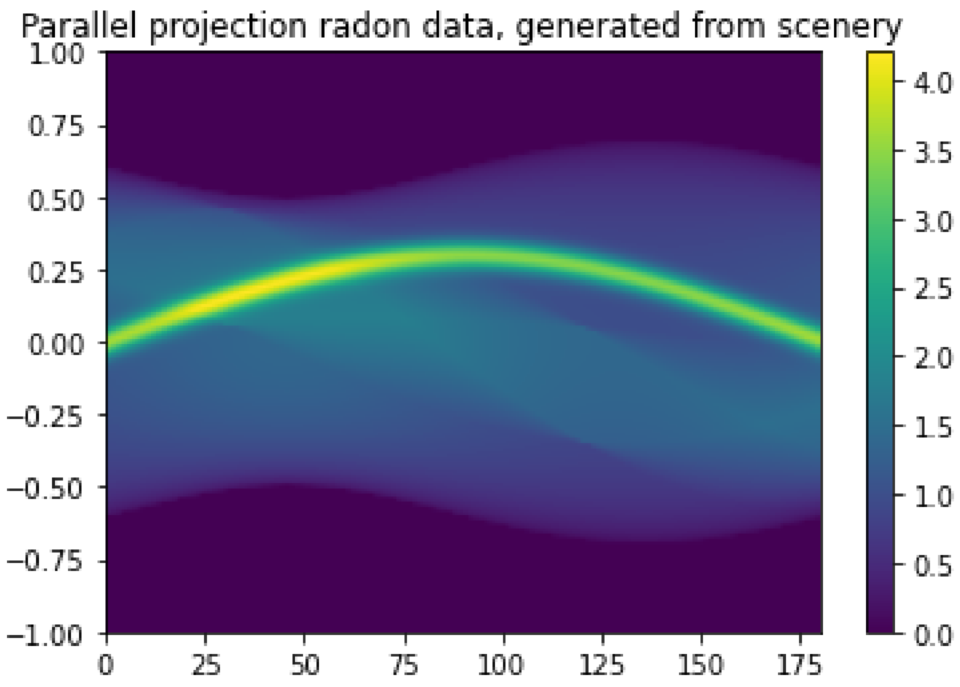
Example: Compare data of parproj and parproj_canvas. Should be roughly the same. Compare the absolute values.

For exponential, use higher values of λ, otherwise the cropping effects will kill the result.

[17]:
```
if radon_interactive:
    scene=[
        ['E',0.5,0.7,0,0,45,1],
        ['R',0.2,0.2,-0.3,0,30,1],
        ['X',-0,-0.3,20,1]
    ]

    N=128
    init_canvas(N,N)
    draw_scene(scene)
    show_canvas()

    p=128
    q=64
    data2=parproj_canvas(p,q)
    data3=parproj(scene,p,q)

    plt.figure()
    plt.imshow(data2,extent=extent,aspect='auto')
    plt.colorbar()
    plt.title('Parallel projection radon data, generated from canvas')
    plt.figure()
    plt.imshow(data3,extent=extent,aspect='auto')
    plt.title('Parallel projection radon data, generated from scenery')
    plt.colorbar()
```



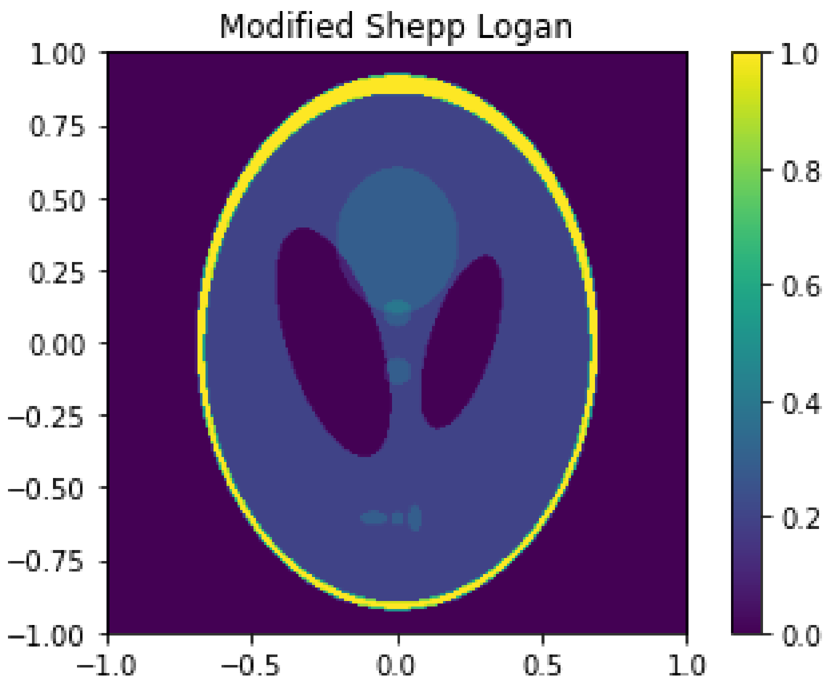




Finally, we can define Shepp Logan and Modified Shepp Logan. We start with the modified.

[18]:
```
ModifiedSheppLogan=[
['E',0.69,0.92,0,0,0,1],
['E',0.6624,0.874,0,-0.0184,0,-0.8],
['E',0.11,0.31,0.22,0,-18,-0.2],
['E',0.16,0.41,-0.22,0,18,-0.2],
['E',0.21,0.25,0,0.35,0,0.1],
['E',0.046,0.046,0,0.1,0,0.1],
['E',0.046,0.046,0,-0.1,0,0.1],
['E',0.046,0.023,-0.08,-0.605,0,0.1],
['E',0.023,0.023,0,-0.606,0,0.1],
['E',0.023,0.046,0.06,-0.605,0,0.1],
]
```

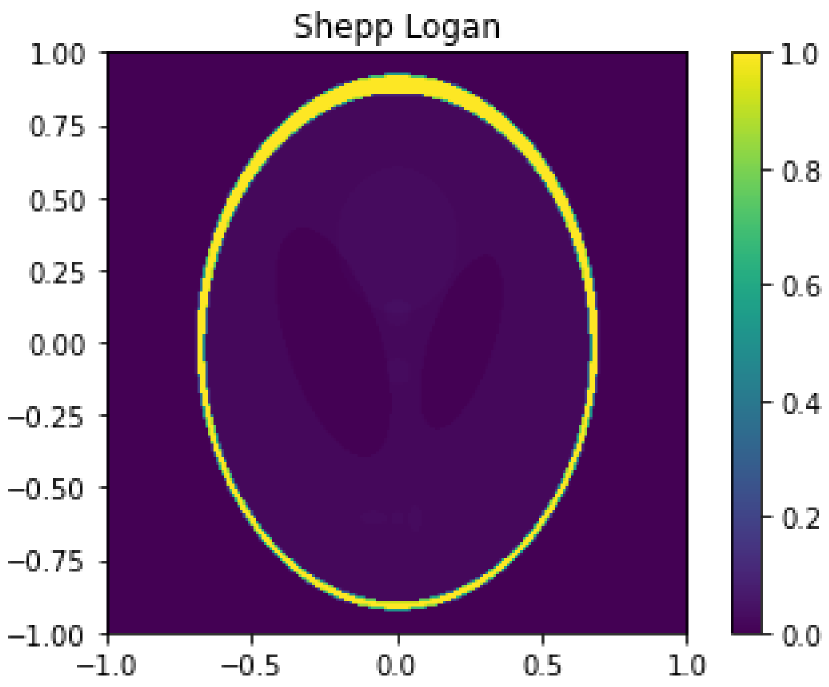
[19]:
```
if radon_interactive:
    init_canvas(128,128)
    draw_scene(ModifiedSheppLogan)
    show_canvas()
    plt.title('Modified Shepp Logan')
    plt.colorbar();
```



In fact, the modified image (which is used by matlab by default) is fraud. It does not represent typical medical values. We should use the true phantom.

[20]:
```
SheppLogan=[
['E',0.69,0.92,0,0,0,1],
['E',0.6624,0.874,0,-0.0184,0,-0.98],
['E',0.11,0.31,0.22,0,-18,-0.02],
['E',0.16,0.41,-0.22,0,18,-0.02],
['E',0.21,0.25,0,0.35,0,0.01],
['E',0.046,0.046,0,0.1,0,0.01],
['E',0.046,0.046,0,-0.1,0,0.01],
['E',0.046,0.023,-0.08,-0.605,0,0.01],
['E',0.023,0.023,0,-0.606,0,0.01],
['E',0.023,0.046,0.06,-0.605,0,0.01],
]
```

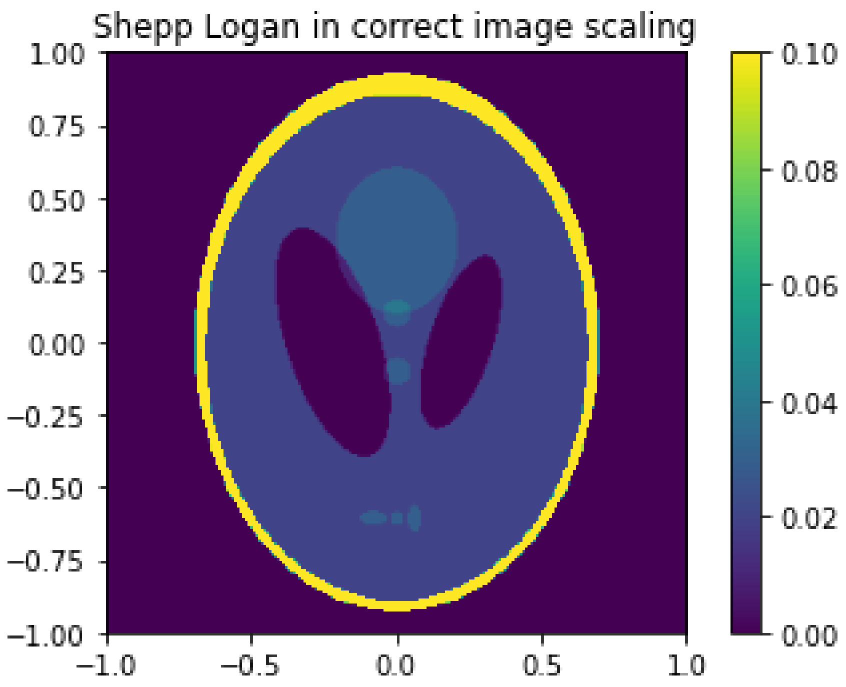
[21]:
```
if radon_interactive:
    init_canvas(128,128)
    draw_scene(SheppLogan)
    show_canvas()
    plt.title('Shepp Logan')
    plt.colorbar();
```



You can easily spot the problem: contrasts in medical tomography are much smaller than in the modified phantom, which makes reconstruction **a lot** more difficult than for the now commonly used modified phantom.

The true phantom models a head, which is surrounded by the skull (**big** contrast) and has soft inner tissue with very little contrast. To spot the challenges, we must tune the colorbar.

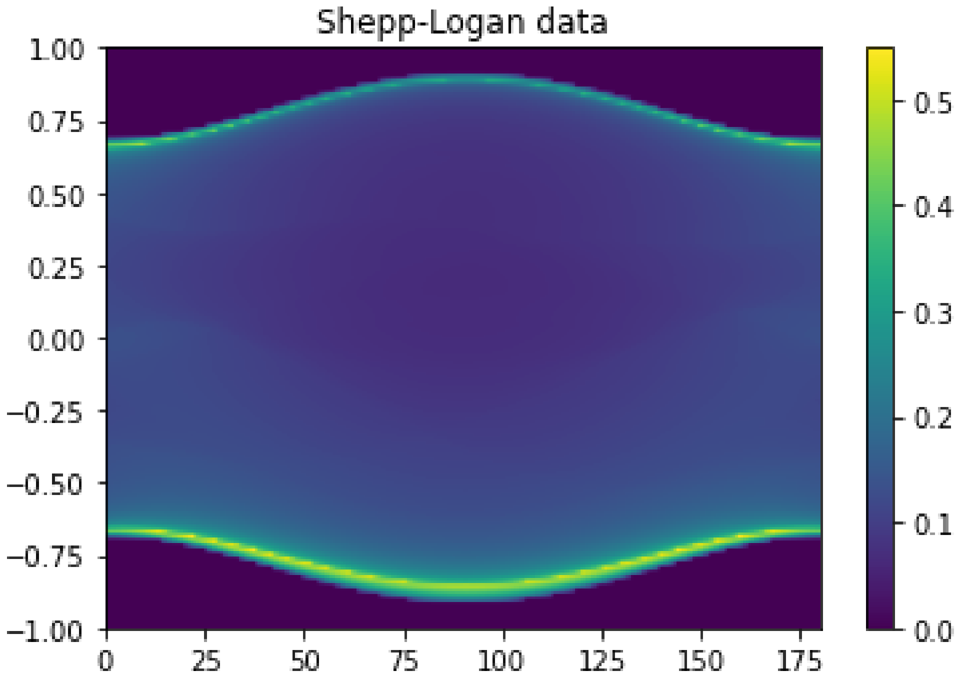
[22]:
```
if radon_interactive:
    init_canvas(128,128)
    draw_scene(SheppLogan)
    img=show_canvas()
    plt.title('Shepp Logan in correct image scaling')
    plt.colorbar();
    img.set_clim(0,0.1)
```

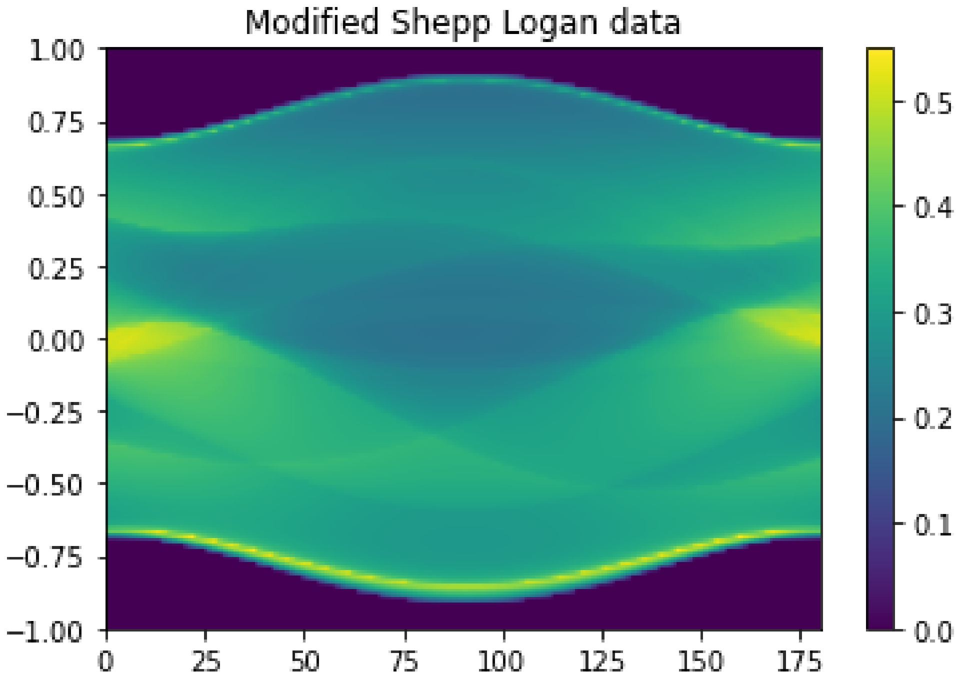


Here's what to look for in a good reconstruction: Are the small objects at the lower end reconstructed and separated? Do the large objects cause artefacts? Are the small objects in the center separated from their larger neighbors?

Now here's finally the Radon data we will work with. Note that we start with very moderate values of p and q.

[23]:
```
if radon_interactive:
    p=128
    q=64
    init_canvas(128,128)
    draw_scene(SheppLogan)
    SheppLoganImage=canvas
    #data_shepplogan=parproj_img(shepplogan,p,q)
    data_SheppLogan=parproj(SheppLogan,p,q)
    clear_canvas()
    draw_scene(ModifiedSheppLogan)
    ModifiedSheppLoganImage=canvas
    data_ModifiedSheppLogan=parproj(ModifiedSheppLogan,p,q)
    extent=[0,180,-1,1]
    plt.title('Shepp-Logan data')
    plt.imshow(data_SheppLogan,extent=extent,aspect='auto')
    plt.colorbar()
    plt.figure()
    plt.title('Modified Shepp Logan data')
    plt.imshow(data_ModifiedSheppLogan,extent=extent,aspect='auto')
    plt.colorbar();
```

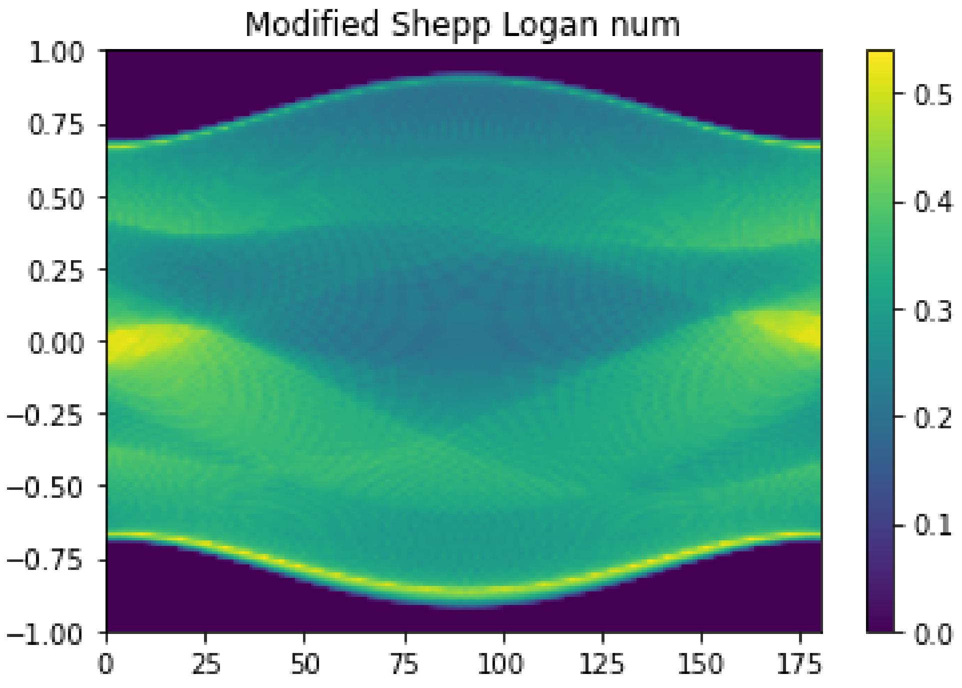


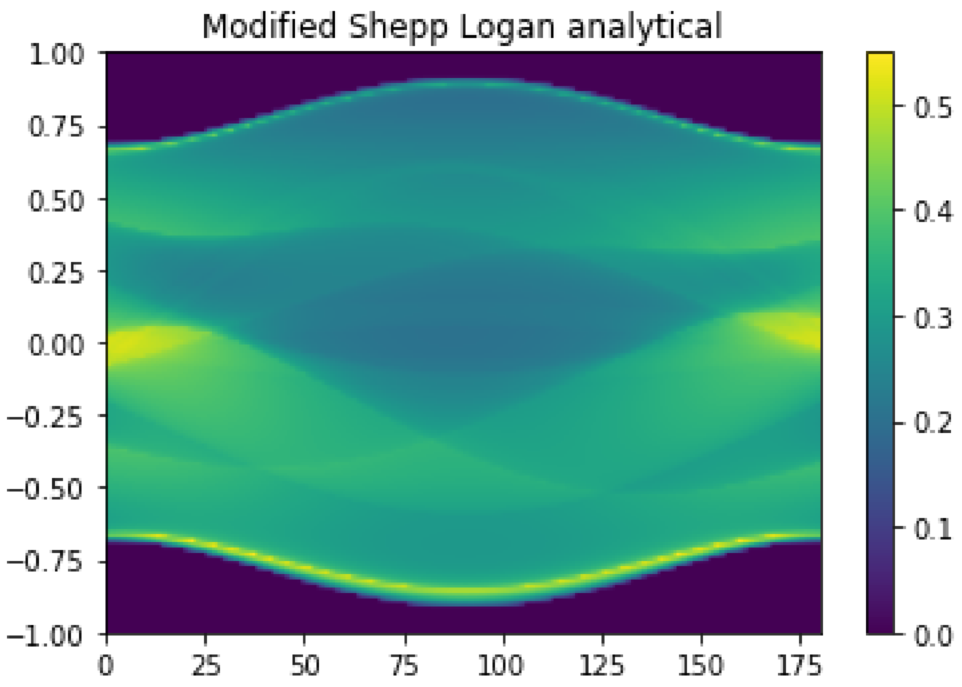


Again, note that in the modified set, objects can be identified, while in the true data set they are almost invisible.

For reference, we compute the data numerically.

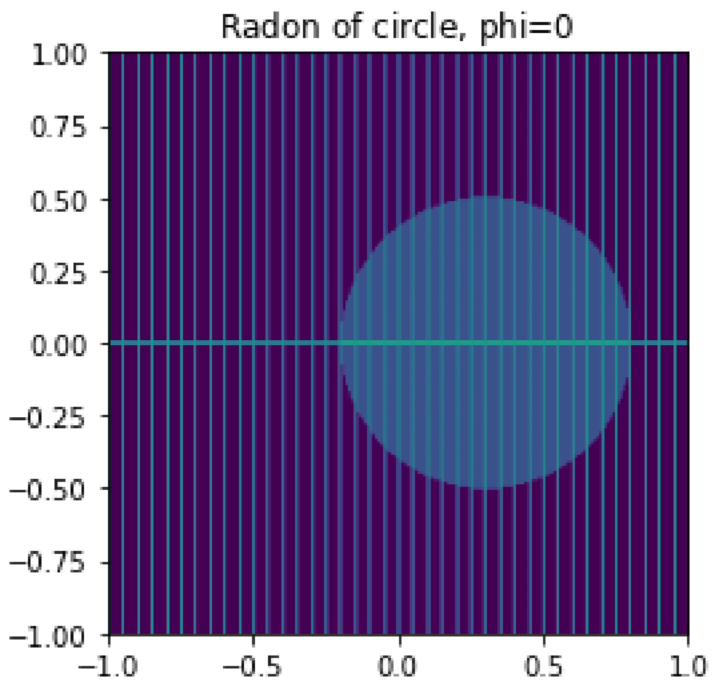
```
[24]: if radon_interactive:
          N=512
          init_canvas(N,N)
          draw_scene(ModifiedSheppLogan)
          data_ModifiedSheppLogan_num=parproj_canvas(p,q)
          plt.title('Modified Shepp Logan num')
          plt.imshow(data_ModifiedSheppLogan_num,extent=extent,aspect='auto')
          plt.colorbar();
          plt.figure()
          plt.title('Modified Shepp Logan analytical')
          plt.imshow(data_ModifiedSheppLogan,extent=extent,aspect='auto')
          plt.colorbar();
```

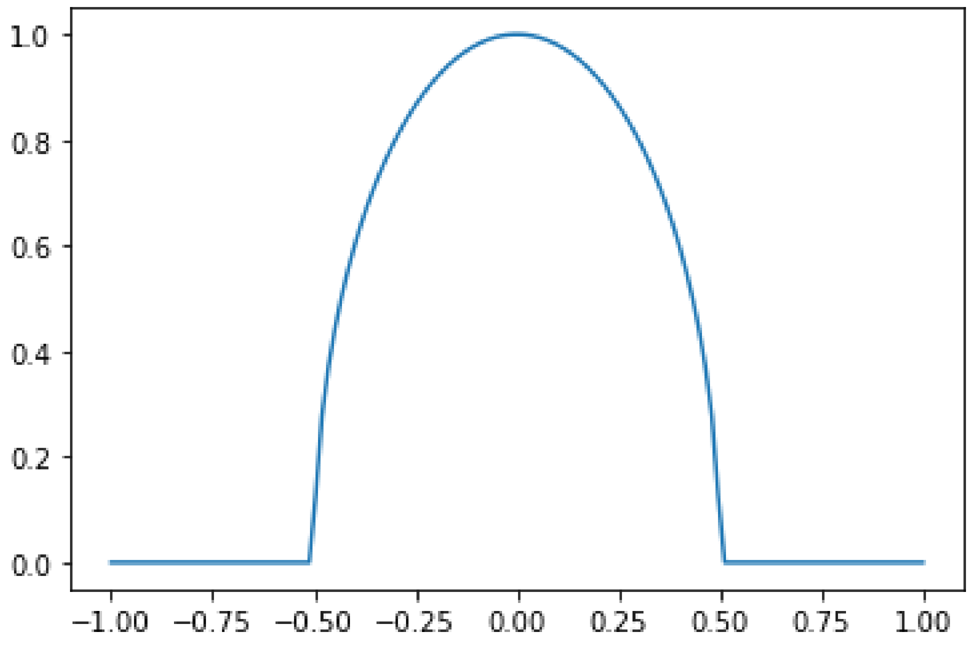




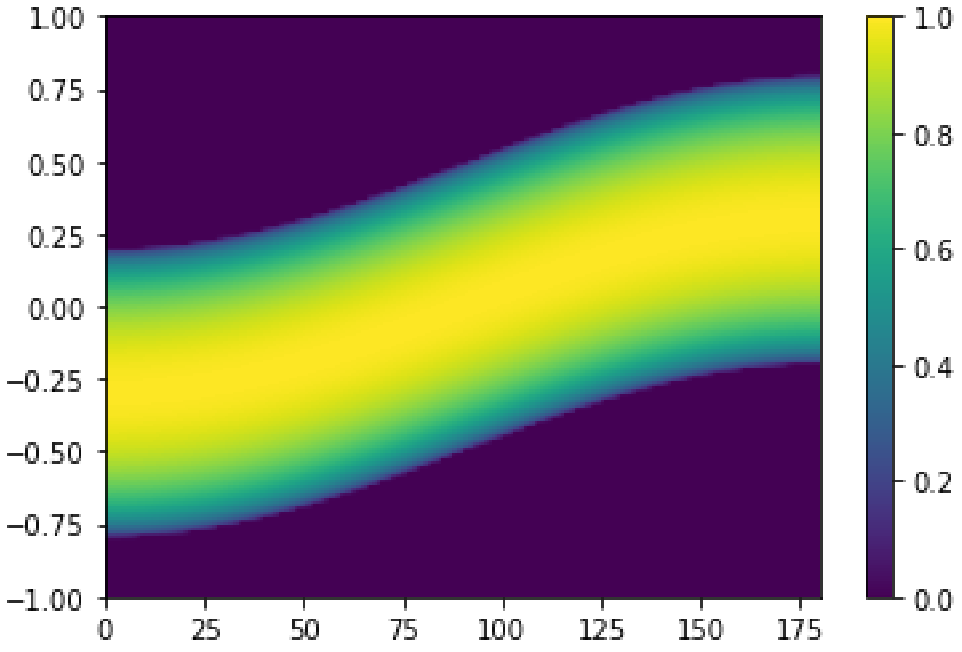
2 Examples for the lecture

[25]:
```
if radon_interactive:
    N=128
    init_canvas(N,N)
    scene=[['E',0.5,0.5,0.3,0,0,1]]
    draw_scene(scene)
    phi=math.pi/8*0
    draw_line(phi+math.pi/2,0,value=4)
    M=20
    for i in range(-M,M):
        draw_line(phi,i/M,value=2)
    show_canvas()
    plt.title('Radon of circle, phi=$0$');
    plt.figure()
    X=np.linspace(-1,1,N)
    Y=0.5*0.5-X*X
    Y[np.where(Y<=0)]=0
    Y=2*np.sqrt(Y)
    plt.plot(X,Y)
```





[26]:
```
if radon_interactive:
    p=128
    q=64
    data=parproj(scene,p,q)
    plt.imshow(data,extent=[0,180,-1,1],aspect='auto')
    plt.colorbar()
```



Weihnachtsaufgabe

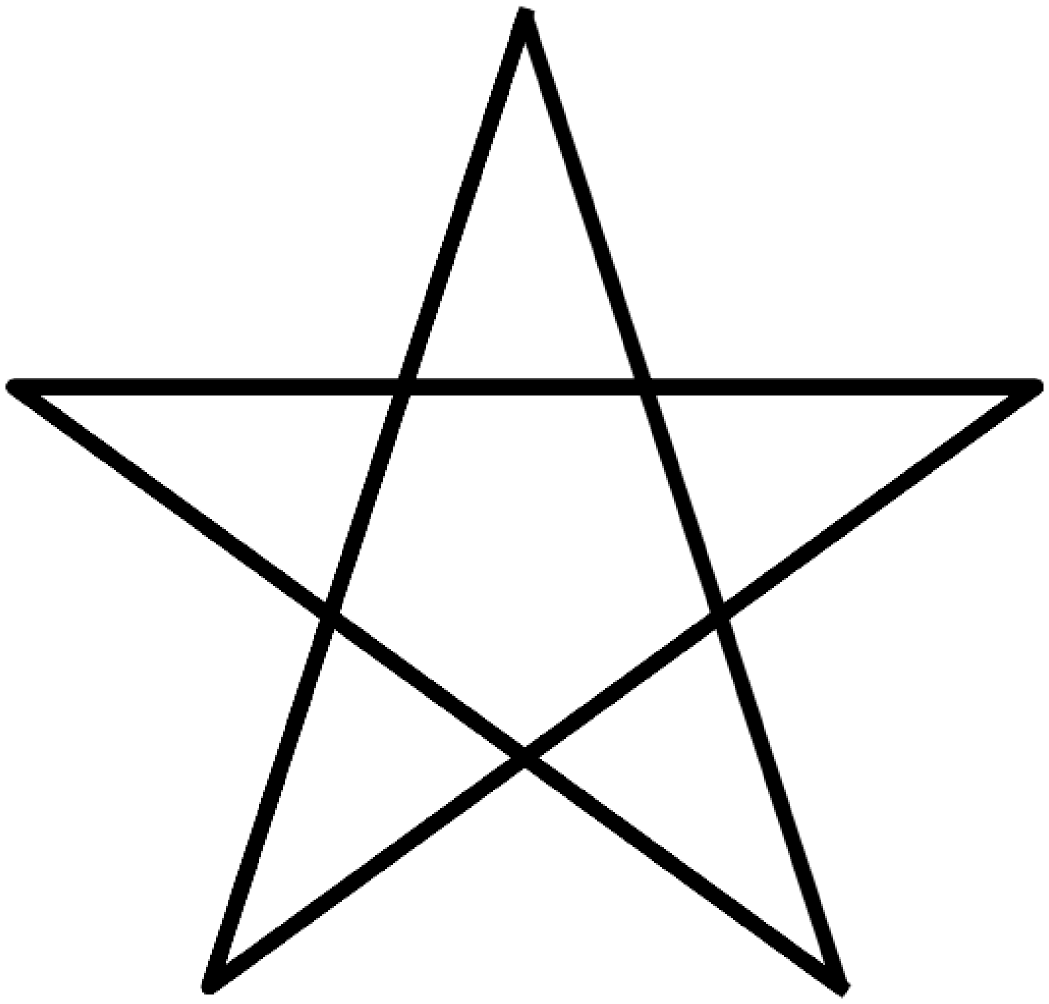
January 31, 2021

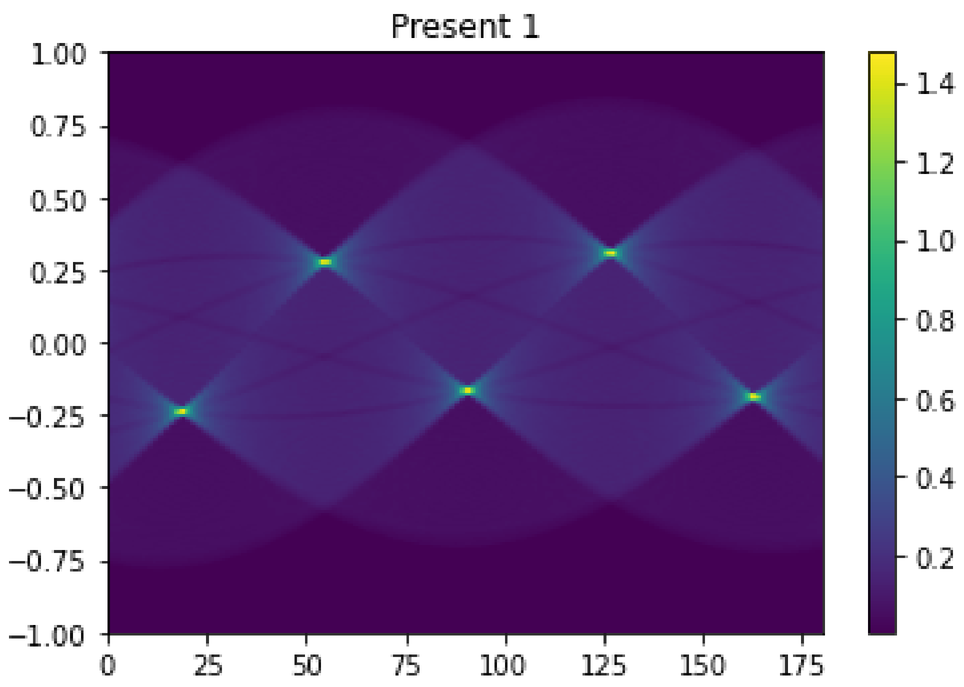
1 Weihnachtsaufgabe inverse Probleme

[6]:
```
bp_interactive=False
%run backprojection.ipynb
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
```

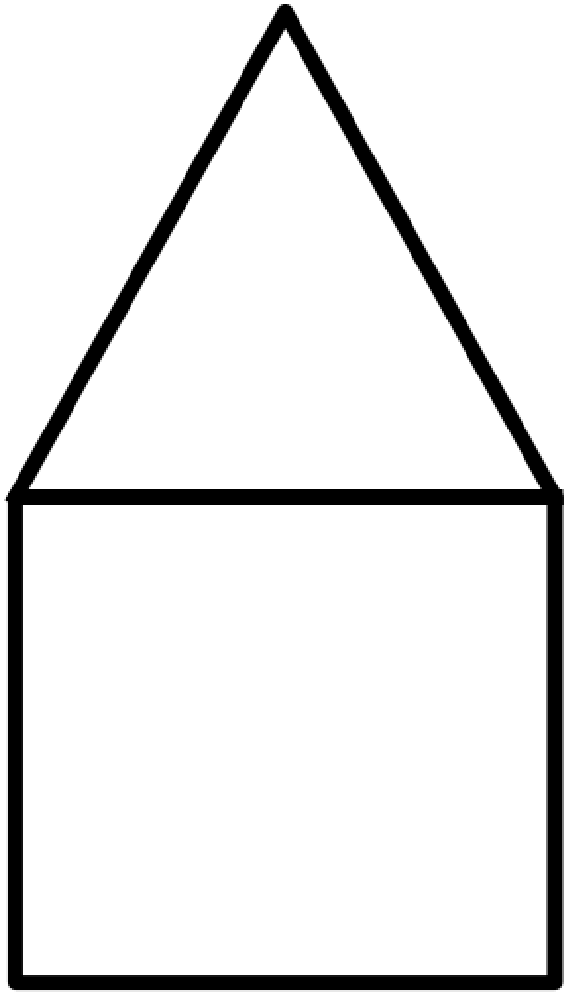
[6]:
```
def get_bitmap():
    fig = plt.gcf()
    canvas = FigureCanvas(fig)
    ax = fig.gca()
    canvas.draw()       # draw the canvas, cache the renderer
    bitmap = np.frombuffer(canvas.tostring_rgb(), dtype='uint8')
    width, height = fig.get_size_inches() * fig.get_dpi()
    bitmap = bitmap.reshape(int(height), int(width), 3)
    bitmap=bitmap[:,:,2]
    N=bitmap.shape[0]
    bitmap=bitmap[0:N-1,0:N-1]
    return bitmap,N//2-1

def draw_sinogram(M):
    global canvas
    bitmap,N=get_bitmap()
    init_canvas(N)
    canvas=1-bitmap.astype(float)/256

    plt.figure()
    data=parproj_canvas(p=128,q=128);
    extent=[0,180,-1,1]
    plt.imshow(data,extent=extent,aspect='auto')
    plt.colorbar()
    plt.title('Present '+str(M))
    plt.imsave('data'+str(M)+'.jpg', data)
    return data
```

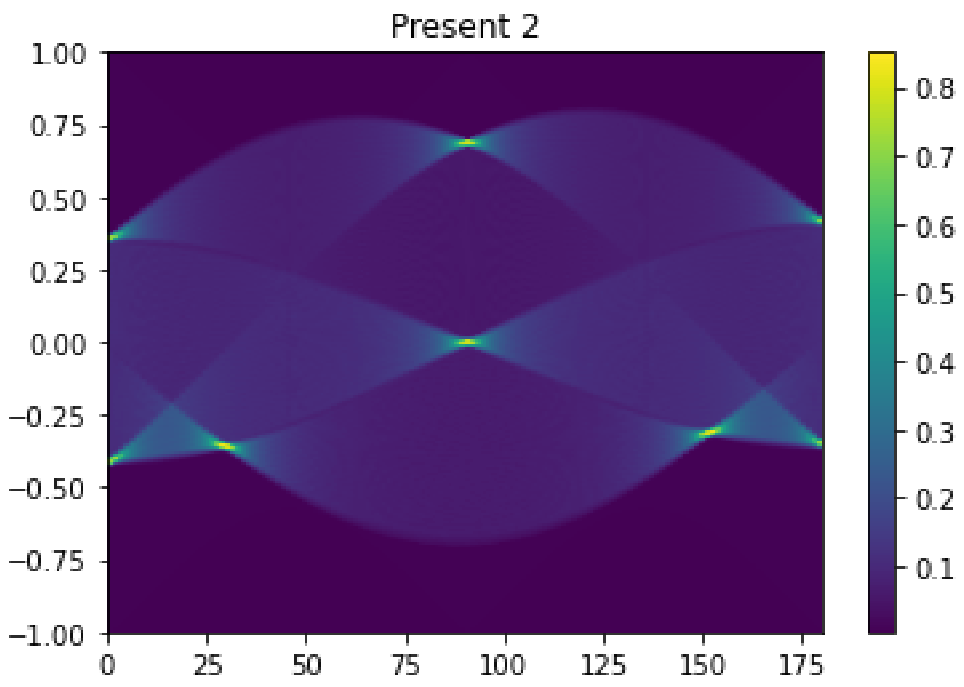
[7]:
```
plt.figure(figsize=(10,10))
theta=np.arange(0,10)*4/5*math.pi+math.pi/2
X=np.cos(theta);
Y=np.sin(theta);
plt.plot(X,Y,color='black',linewidth=8)
#plt.plot([0.5 0 -0.5],[0 0.9 0])
plt.axis('off')
plt.axis('equal')
draw_sinogram(1);
```





```
[8]: plt.figure(figsize=(10,10))
     # In einem Zug!
     X=[ 0.5, -0.5, -0.5, -0.5, 0.5,0.5,0,-0.5]
     Y=[ 0, 0, -0.9,-0.9, -0.9,0,0.9,0]
     plt.plot(X,Y,color='black',linewidth=8)
     #plt.plot([0.5 0 -0.5],[0 0.9 0])
     plt.axis('off')
     plt.axis('equal')
     draw_sinogram(2);
```





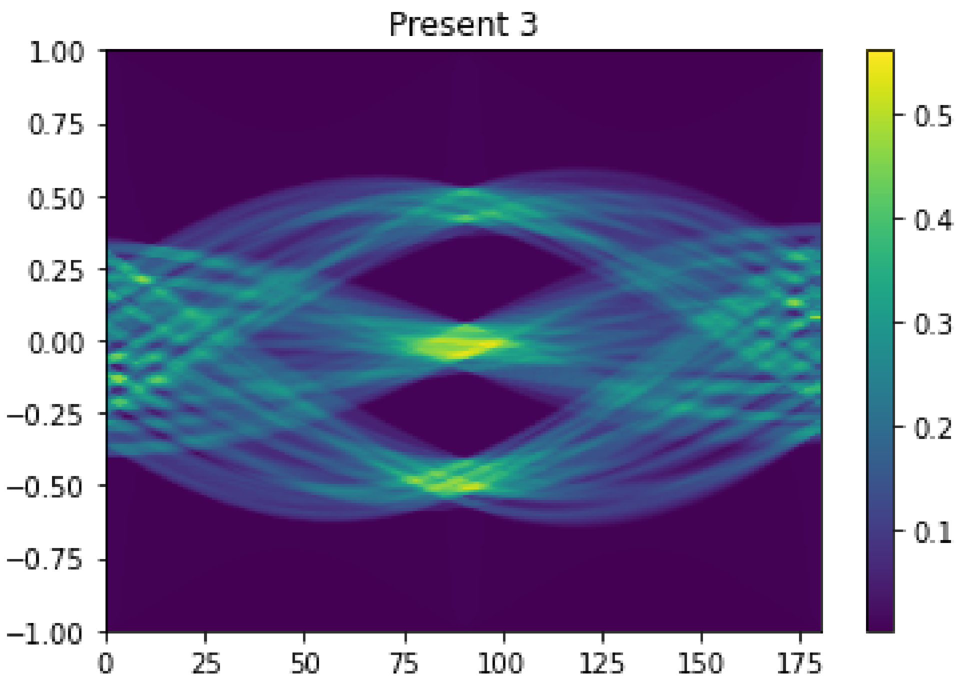
[9]:
```
plt.figure(figsize=(10,10))

plt.text(0,0.
 ↪6,'Frohe',verticalalignment='center',horizontalalignment='center',fontsize=80,fontweight='bol
plt.
 ↪text(0,0,'Weihn',verticalalignment='center',horizontalalignment='center',fontsize=80,fontweig
plt.text(0,-0.
 ↪6,'acht',verticalalignment='center',horizontalalignment='center',fontsize=80,fontweight='bold
plt.axis('off')
plt.axis('equal')
plt.xlim([-1,1])
plt.ylim([-1,1])

data=draw_sinogram(3)
plt.figure()
show_canvas()
```

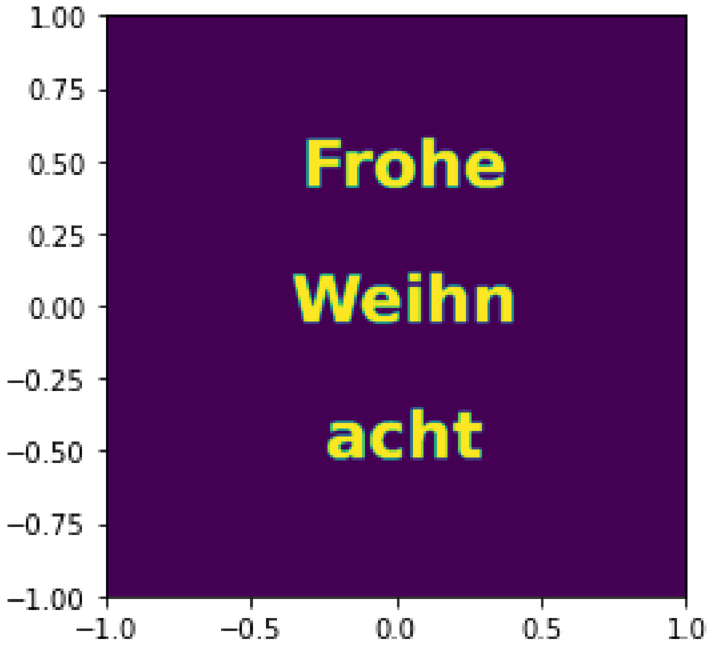
[9]:
```
<matplotlib.image.AxesImage at 0x7f3da62a8b80>
```

Frohe

Weihn

acht

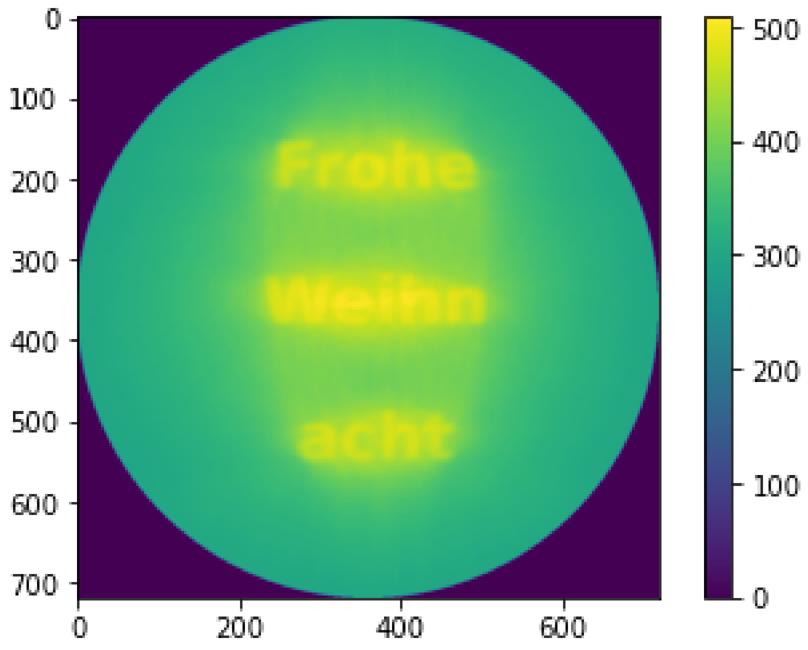




[6]:
```
dataneu=np.array(plt.imread('data3.jpg')[:,:])
dataneu=np.sum(dataneu,axis=2)
print(data.shape)
print(dataneu.shape)
imgneu=backproj(dataneu)
plt.imshow(imgneu)
#plt.clim([500,700])
plt.colorbar()
```

```
(257, 128)
(257, 128)
```

```
[6]: <matplotlib.colorbar.Colorbar at 0x7feb45c86040>
```



```
[ ]:
```

backprojection

January 31, 2021

1 Backprojection

Perform unfiltered backprojection.

```
[2]: radon_interactive=False
     %run "../05 Radon Transform/radon.ipynb"
     try: bp_interactive
     except NameError: bp_interactive = True
```

Generate data for a simple test object.

```
[ ]: if (bp_interactive):
         scene=[['E',0.4,0.3,0.4,0.2,30,1]]
         N=128
         init_canvas(N)
         draw_scene(scene)
         show_canvas()
         plt.title('Image')
         plt.colorbar()
         ellipse_image=canvas
         p=128
         q=128
         ellipse_data=parproj(scene,p,q)
         plt.figure()
         extent=[0,180,-1,1]
         plt.imshow(ellipse_data,extent=extent,aspect='auto')
         plt.title('Data')
         plt.colorbar();
```

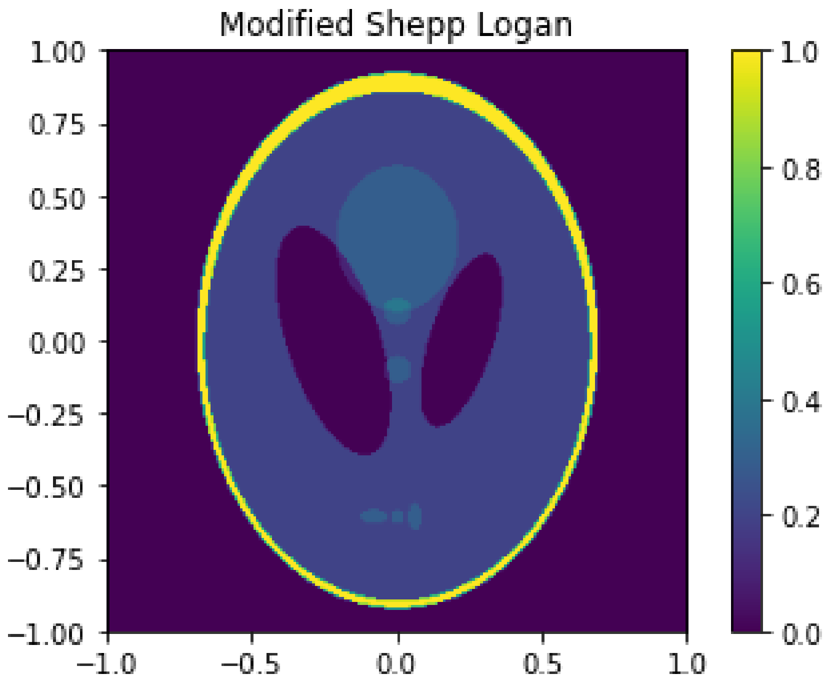
```
[ ]: def backproj(data,p=None,phivec=None):
         global canvas
         q,p0=data.shape
         if (p==None):
             p=p0
         q=(q-1)//2
         if (phivec is None):
             phivec=np.arange(0,p)*math.pi/p
```

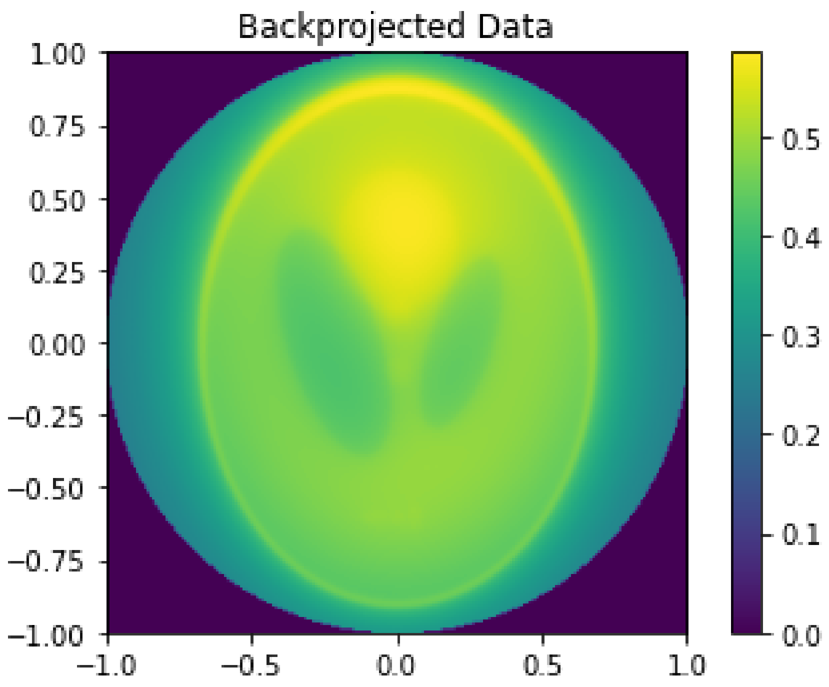
```
    p=phivec.size

    hq=2/(2*q+1)
    Xq=np.arange(0,2*q+1)*hq-1+hq/2
    imgsum=np.zeros(canvas_X.shape)
    for i,phi in enumerate(phivec):
        v=data[:,i]
        proj=canvas_X*math.cos(phi)+canvas_Y*math.sin(phi)
        img=scipy.interpolate.interpn([Xq],v,proj.
 ↪flat,bounds_error=False,fill_value=0)
        imgsum=imgsum+np.reshape(img,canvas_X.shape)
    saveimg=canvas
    clear_canvas()
    draw_ball()
    imgsum=imgsum*canvas/p*math.pi/2
    canvas=saveimg
    return imgsum
```

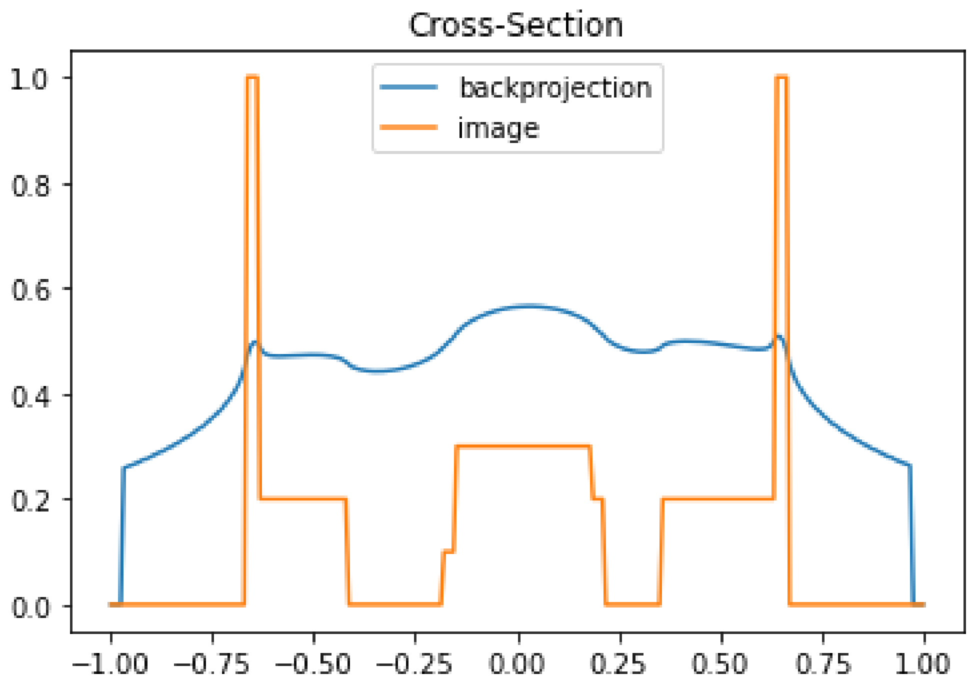
```
[ ]: if bp_interactive:
         phivec=np.arange(0,p)*math.pi/p
         slice=1
         data0=np.array([ellipse_data[:,slice]]).transpose()
         plt.plot(data0)
         plt.title('Data for $\\varphi=$'+format(phivec[slice]*180/math.pi,'3.1f'))
         plt.figure()

         canvas=backproj(data0,phivec=np.array([phivec[slice]]))
         show_canvas()
         plt.title('Backprojection from angle phi')
         plt.colorbar()
         plt.figure()

         canvas=backproj(ellipse_data)
         show_canvas()
         plt.title('Backprojection from all angles')
         plt.colorbar()
         plt.figure()

         X=canvas_X[1,:]
         slice=N-30
         plt.plot(X,canvas[slice,:],X,ellipse_image[slice,:])
         plt.title('Cross-Section')
         plt.legend(['backprojection','image'])
```

```
[9]: if (bp_interactive):
         scene=ModifiedSheppLogan
         clear_canvas()
```

```
draw_scene(scene)
show_canvas()
plt.title('Modified Shepp Logan')
plt.colorbar()
image=canvas
plt.figure()
data=parproj(scene,p,q)
plt.imshow(data,extent=extent,aspect='auto')
canvas=backproj(data)
show_canvas()
plt.colorbar()
plt.title('Backprojected Data')
plt.figure()

X=canvas_X[1,:]
slice=N-30
plt.plot(X,canvas[slice,:],X,image[slice,:])
plt.title('Cross-Section')
plt.legend(['backprojection','image'])
```

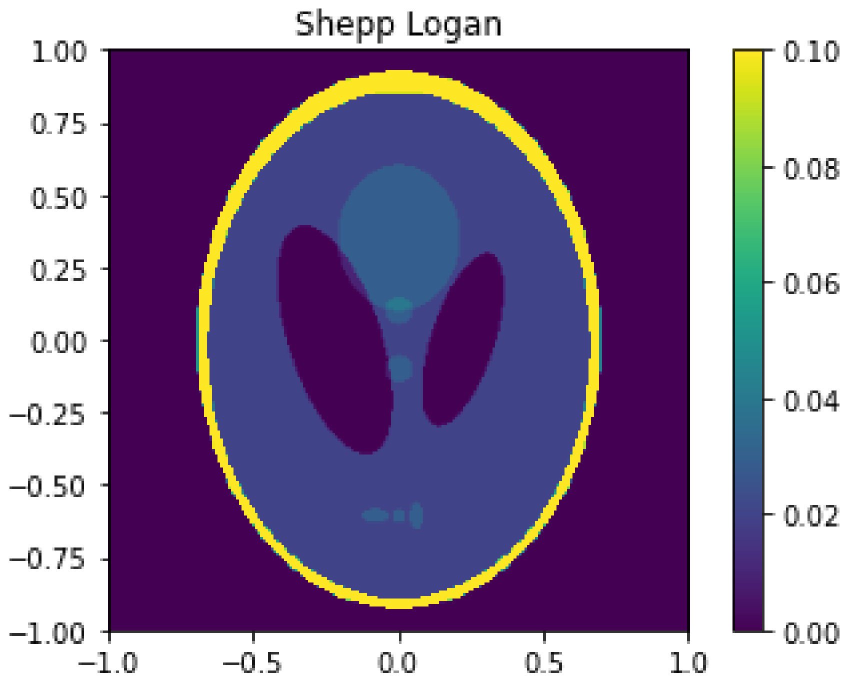


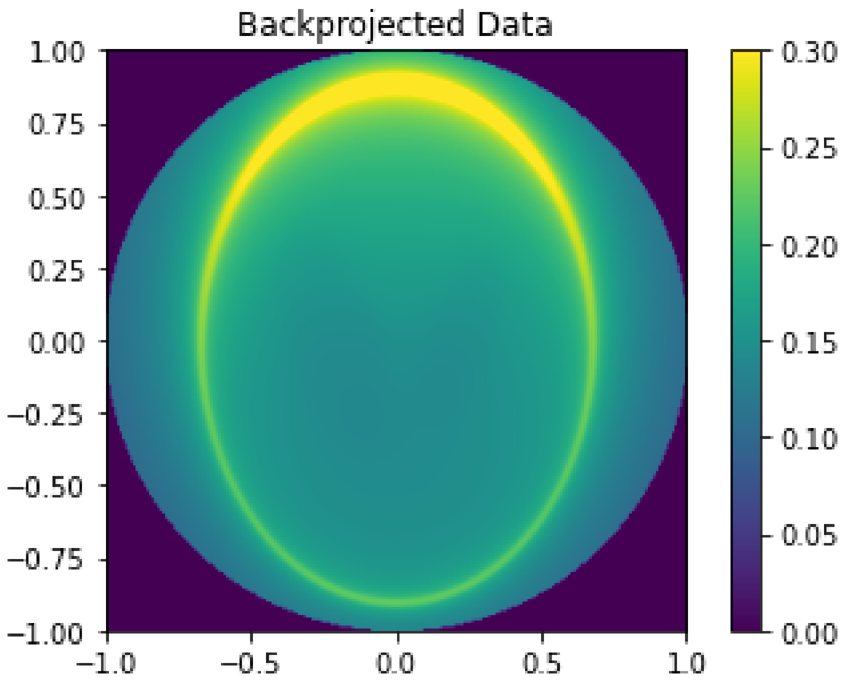


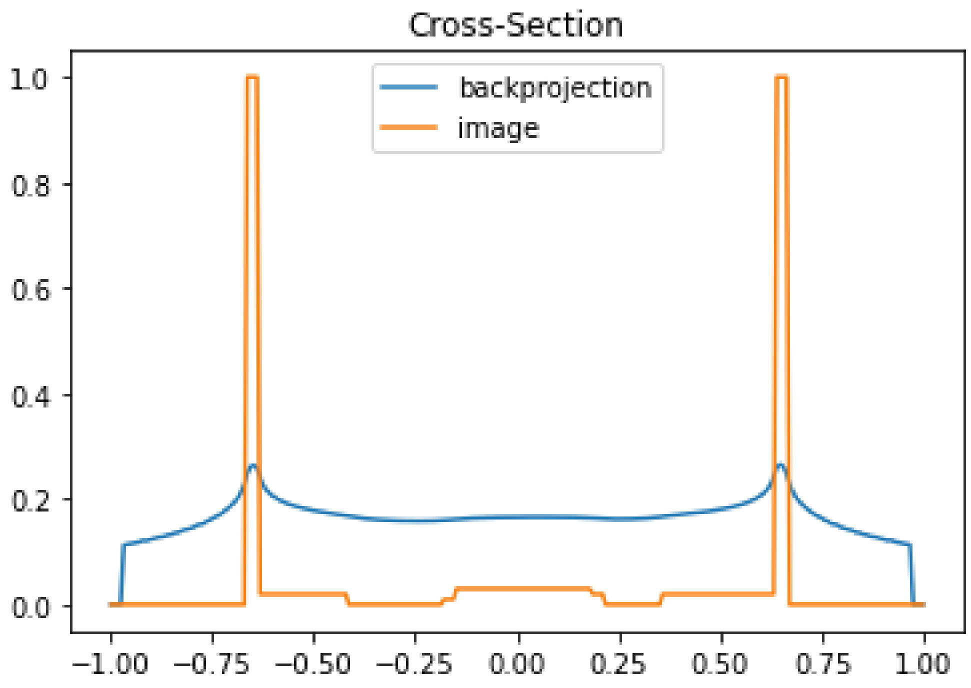


[10]:
```
if (bp_interactive):
    scene=SheppLogan
```

```
clear_canvas()
draw_scene(scene)
plot=show_canvas()
plot.set_clim(0,0.1)
plt.title('Shepp Logan')
plt.colorbar()
image=canvas
plt.figure()
data=parproj(scene,p,q)
plt.imshow(data,extent=extent,aspect='auto')
canvas=backproj(data)
plot=show_canvas()
plot.set_clim(0,0.3)
plt.colorbar()
plt.title('Backprojected Data')
plt.figure()

X=canvas_X[1,:]
slice=N-30
plt.plot(X,canvas[slice,:],X,image[slice,:])
plt.title('Cross-Section')
plt.legend(['backprojection','image'])
```




Backprojected Data
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0
0.30
0.25
0.20
0.15
0.10
0.05
0.00


Cross-Section
backprojection
image
1.0
0.8
0.6
0.4
0.2
0.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

```
[ ]:
```

Bandlimit

January 31, 2021

1 Bandlimited functions

Let $F \in \mathcal{S}$. Let X_Ω the band limited functions with band limit Ω, that is

$$X_\Omega = \{f \in \mathcal{S} : |\widehat{f}(\xi)| = 0, ||\xi|| > |\Omega|.\}$$

Then due to Parseval, the best approximation to F in X_Ω is given by

$$(\widehat{F} \cdot \widetilde{\chi_{[-\Omega,\Omega]}}) = \frac{1}{\sqrt{2\pi}} f(x) * \Omega\sqrt{\frac{2}{\pi}} \text{sinc}\,(\Omega x) = \frac{\Omega}{\pi} f * \text{sinc}(\Omega \cdot).$$

For periodic functions and bandlimited Fourier series, we have a corresponding theorem (interpreting all in the distributional sense). We look at the example $F = \chi_{[-1,1]}$ and use the integral sinc function. Note that we have

$$Si(x) = \int_0^x \text{sinc}(t)\, dt = \Omega \int_0^{x/\Omega} \text{sinc}\ (\Omega y)\ dy$$

and thus

$$\int_0^x \text{sinc}(\Omega y)\, dy = \frac{1}{\Omega} Si(\Omega x).$$

So for our example, we have that the best approximation to F is given by

$$\frac{1}{\pi}(Si(\Omega(x+1)) - Si(\Omega(x-1))).$$

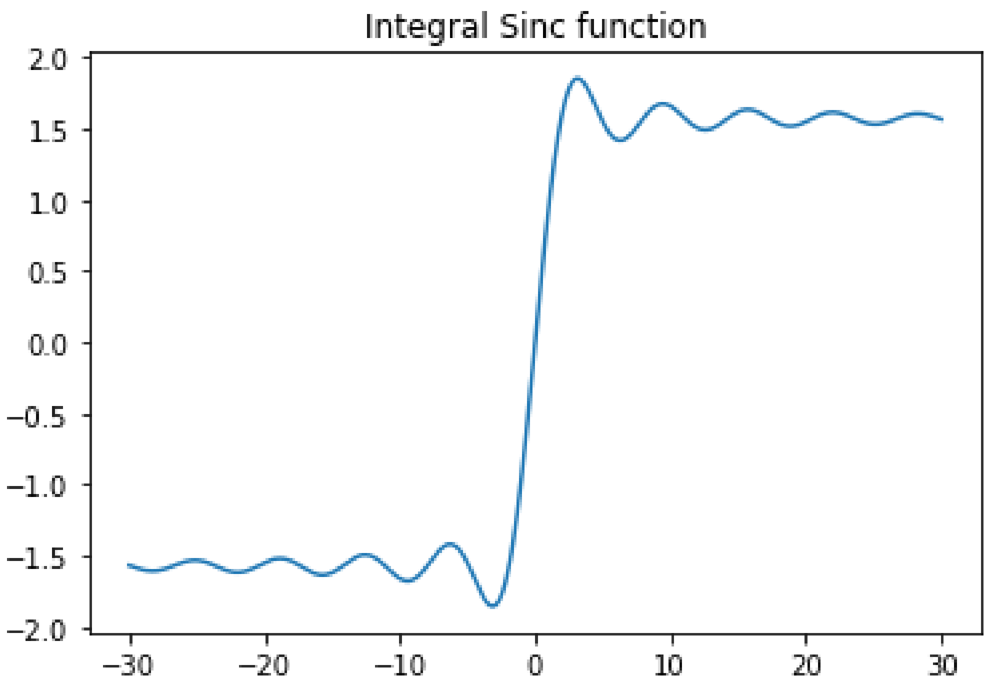
Also note that python has the sinc defined with π, but Si is not defined with π! Weird.

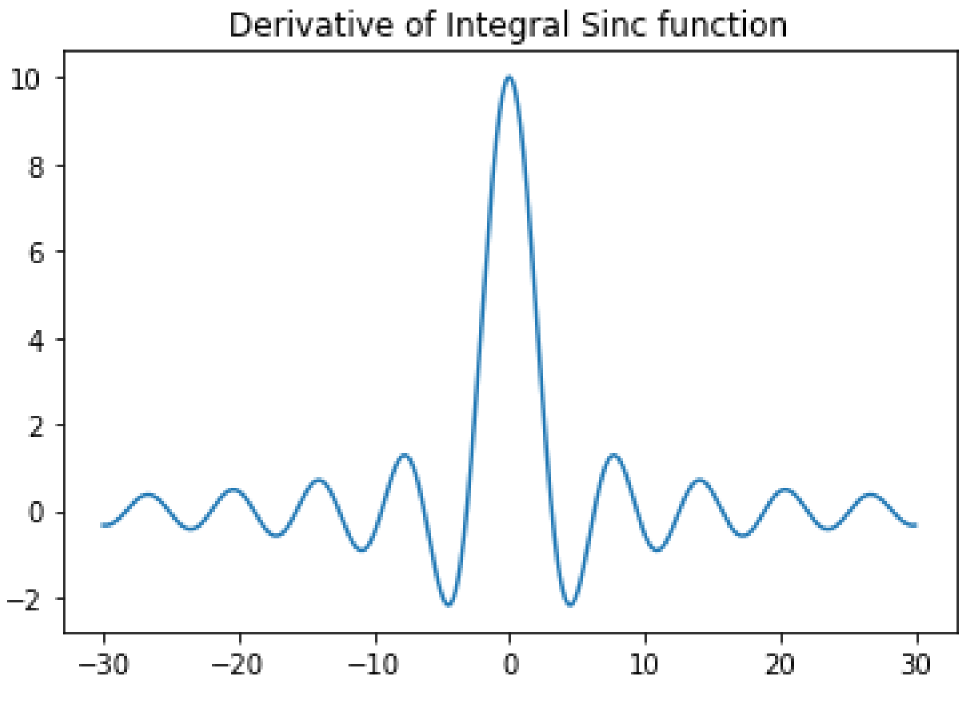
Note that the best approximation is not a best approximation in the C^∞-sense due to the Gibbs-effect!

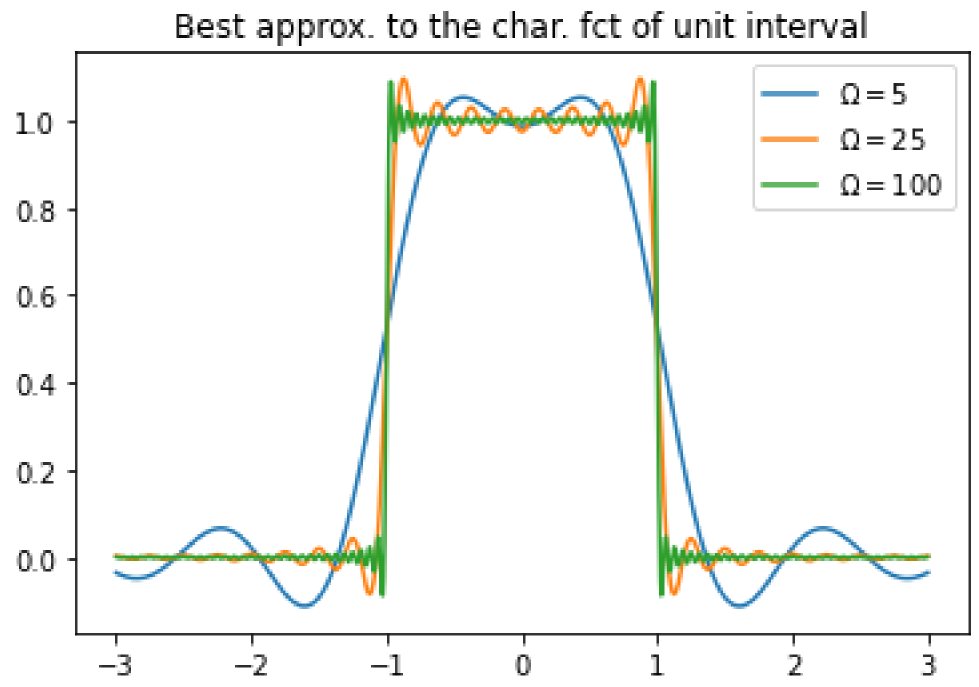
[1]:
```
import numpy as np
import math
import scipy
import scipy.special
import matplotlib.pyplot as plt
```

[2]:
```
#%matplotlib widget
%matplotlib inline
def f(x):
    x=np.array(x)
    I=np.where(abs(x)<1)
    y=np.zeros(x.shape)
    y[I]=1
    return y
def best(x,Omega):
    (si,ci)=scipy.special.sici(Omega*(x+1))
    (si1,ci)=scipy.special.sici(Omega*(x-1))
    # Divide by $\pi$ according to the derivation.
    return (si-si1)/math.pi

N=1024
x=np.linspace(-3,3,N)
#plt.plot(x,f(x))
#plt.figure()
X=10*x
(si,ci)=scipy.special.sici(X)
plt.plot(X,si)
plt.title('Integral Sinc function')
plt.figure()
h=6/N
plt.plot(X[:-1],(si[1:]-si[:-1])/h)
plt.title('Derivative of Integral Sinc function')
plt.figure()
plt.plot(x,best(x,5))
plt.plot(x,best(x,25))
plt.plot(x,best(x,100))
plt.title('Best approx. to the char. fct of unit interval')
plt.legend(['$\\Omega=5$','$\\Omega=25$','$\\Omega=100$'])
```

```
[2]: <matplotlib.legend.Legend at 0x7fcc3e095490>
```


Integral Sinc function
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−30
−20
−10
0
10
20
30


Derivative of Integral Sinc function
10
8
6
4
2
0
−2
−30
−20
−10
0
10
20
30

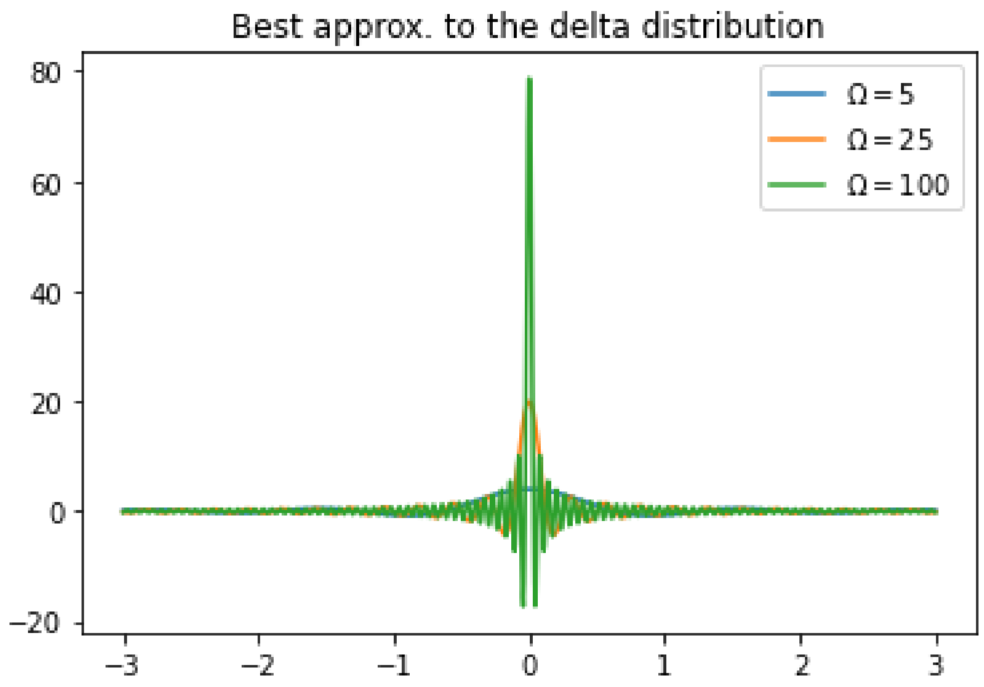


Second example: Delta-Distribution. Since $\widehat{\delta} = 1$, its best approximation in X_Ω is $\chi_{[-\Omega,\Omega]}$. The inverse Fourier Transform then is

$$\Omega\sqrt{\frac{2}{\pi}}\,\mathrm{sinc}(\Omega x)$$

[3]:
```
def sinc(x):
    return np.sinc(x/math.pi)

def best_delta(x,Omega):
    return Omega*math.sqrt(2/math.pi)*sinc(Omega*x)

plt.figure()
plt.plot(x,best_delta(x,5))
plt.plot(x,best_delta(x,25))
plt.plot(x,best_delta(x,100))
plt.title('Best approx. to the delta distribution')
plt.legend(['$\\Omega=5$','$\\Omega=25$','$\\Omega=100$'])
```

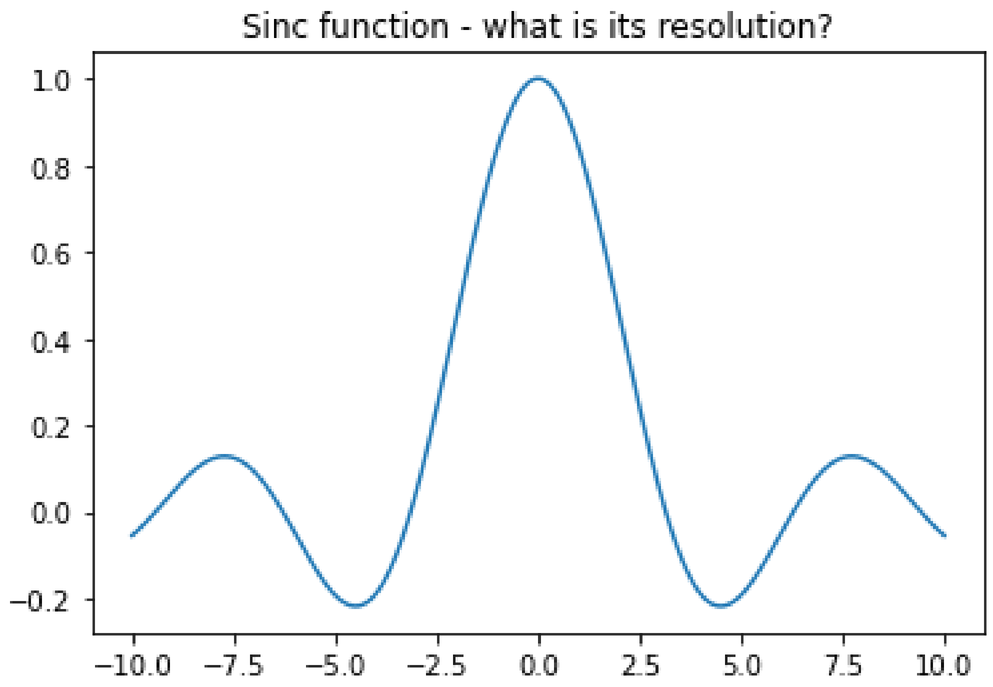
[3]:
```
<matplotlib.legend.Legend at 0x7fcc3defc400>
```



We want to define spatial resolution for bandlimited functions. We set $\Omega = 1$, so a peak at the origin is reconstructed as the sinc function.

```
[4]: x=np.linspace(-10,10,N)
     plt.figure()
     y=sinc(x)
     plt.plot(x,y)
     plt.title('Sinc function - what is its resolution?')
```

```
[4]: Text(0.5, 1.0, 'Sinc function - what is its resolution?')
```

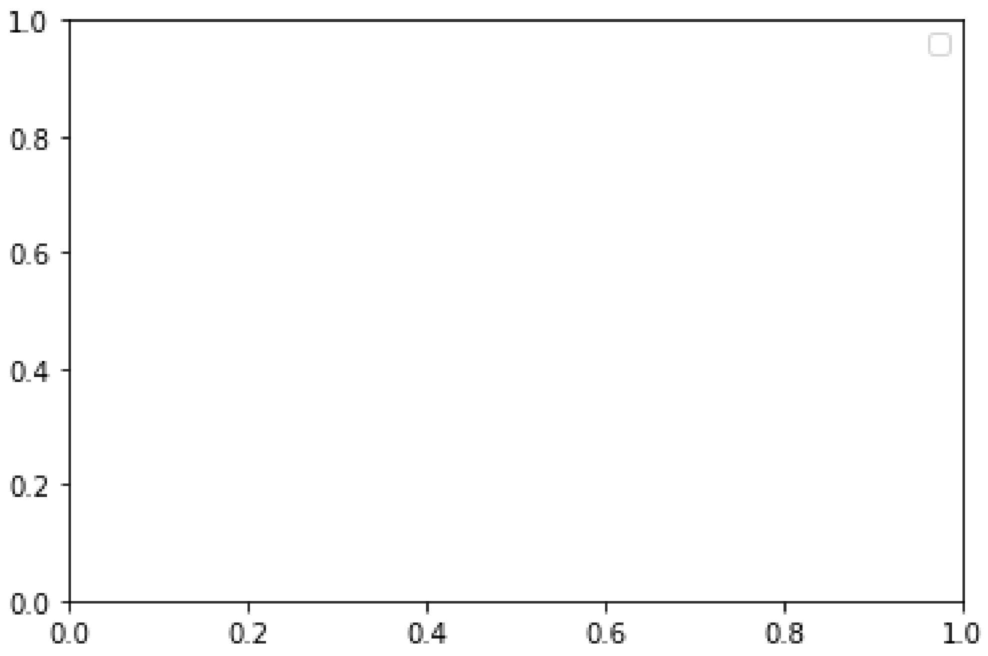


[5]:
```
M=100
from ipywidgets import interactive
def f(z):
    z=z/M*3
    plt.clf()
    Y=sinc(x)+sinc(x+z)
    plt.plot(x,sinc(x-z),x,sinc(x+z))
    plt.plot(x,sinc(x-z)+sinc(x+z))
    return x
plt.figure()
w=interactive(f, z=(0,M))
display(w)
plt.legend(['Sinc functio-n 1','Sinc function 2','Sum'])
```

```
interactive(children=(IntSlider(value=50, description='z'), Output()),␣
 ↪_dom_classes=('widget-interact',))
```

```
<Figure size 432x288 with 0 Axes>
```

[5]:
```
<matplotlib.legend.Legend at 0x7fcc3ddbaa60>
```



FWHM - Full Width Half Maximum

```
[ ]:
```

FBP

January 31, 2021

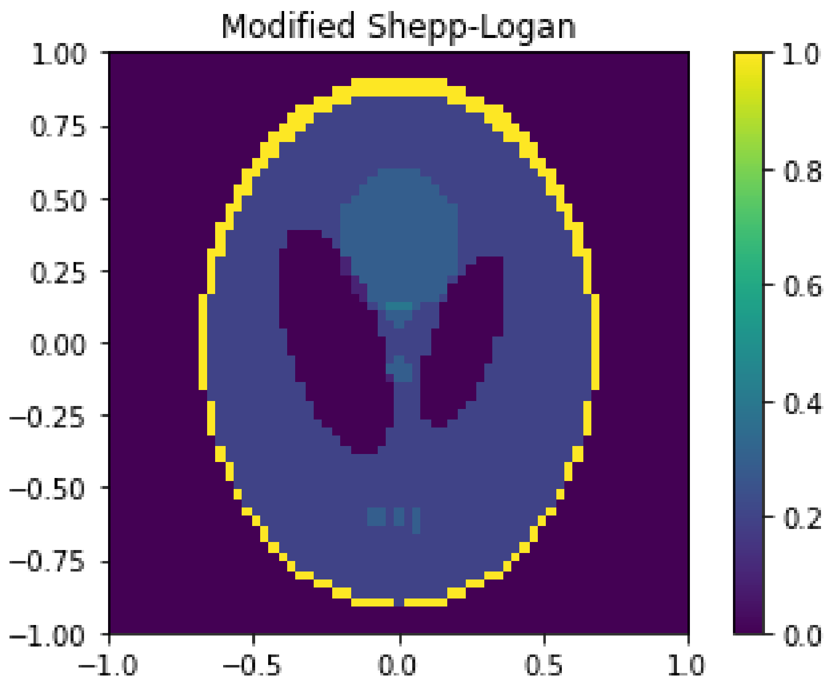
1 Filtered Backprojection - slow implementation according to 6.15

We implement filtered Backprojection according to the summation formula. No interpolation is used, this is an N^4-algorithm. Never use this in production. N=64 takes a couple of minutes.

[1]:
```
bp_interactive=False
%run "../05RadonTransform/backprojection.ipynb"
try: fbp_interactive
except NameError: fbp_interactive = True
```

We generate a standard Modified Shepp Logan phantom.

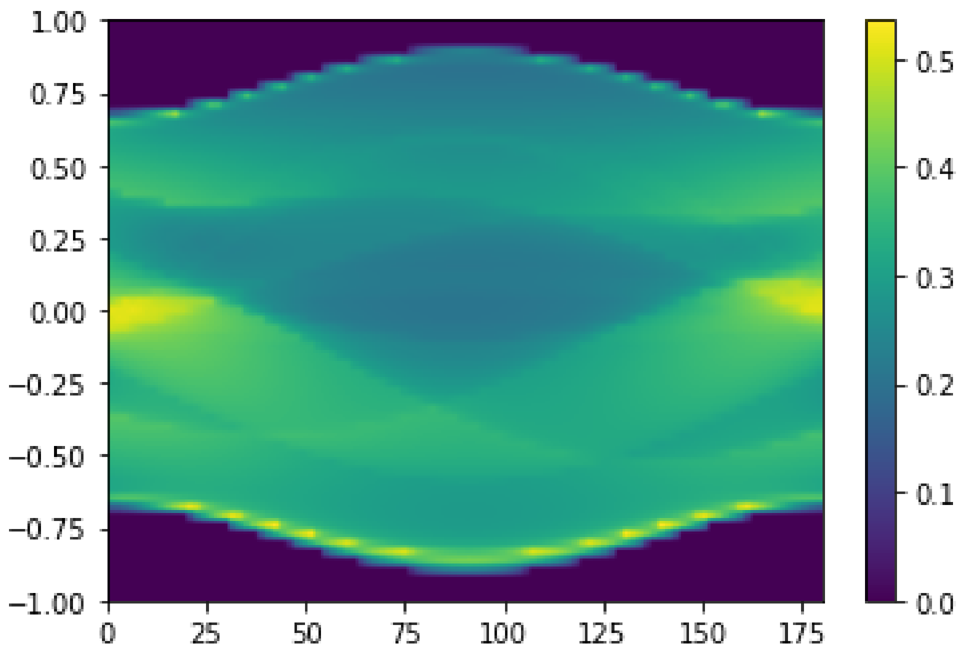
[2]:
```
if (fbp_interactive):
    N=32
    x=np.arange(-N,N+1)
    y=np.arange(-N,N+1)
    Xgrid,Ygrid=np.meshgrid(x,y)
    init_canvas(N,N)
    phantom=ModifiedSheppLogan
    draw_scene(phantom)
    show_canvas()
    plt.colorbar()
    radonextent=[0,180,-1,1]
    plt.title('Modified Shepp-Logan')
```



Generate data. p,q, Ω are chosen in an optimal way (are they? Check).

Note that the data are produced from the original definition of the phantom, not from the pixel phantom! This is different from how the data is usually produced in Matlab.

[3]:
```
if (fbp_interactive):
    q=N
    p=int(np.floor(math.pi*q))
    Omega=p
    phivec=np.arange(0,p)*math.pi/p
    thetavec=np.array((np.cos(phivec),np.sin(phivec))).T
    R=parproj(phantom,p=p,q=q)
    R=parproj(phantom,q=q,phivec=phivec)
    plt.figure()
    plt.imshow(R,extent=radonextent,aspect='auto')
    plt.colorbar()
```

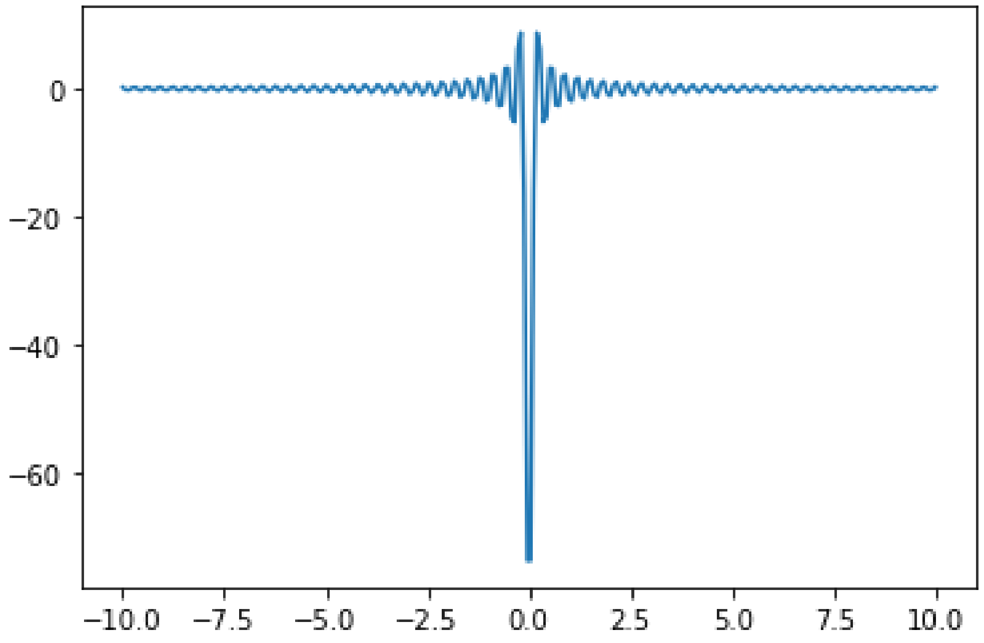


Implement Ram-Lak. Trap:

1. sinc is defined differently in numpy.
2. Evaluating cos(x)-1 is a very bad idea. Use ram_lak2.

[4]:
```
def sinc(s):
    return np.sinc(s/math.pi)
# Ram-Lak hat eine blöde numerische Instabilität
def ram_lak2(s):
    I=np.where(abs(s)<1e-6)
    I1=np.where(abs(s)>=1e-6)
    y=np.zeros(s.shape)
    # Should insert the cos power series. cos(x)=1-x^2/2+x^4/24. But s is small..
 ↪..
    y[I]=Omega*Omega/(4*math.pi*math.pi)*(1/2)
    s1=s[I1]*Omega
    y[I1]=Omega*Omega/(4*math.pi*math.pi)*(sinc(s1)+(np.cos(s1)-1)/(s1*s1))
    return y
def ram_lak(s):
    I=np.where(s==0)
    I1=np.where(s!=0)
    y=np.zeros(s.shape)
    y[I]=Omega*Omega/(8*math.pi*math.pi)
    s1=s[I1]*Omega
    y[I1]=Omega*Omega/(4*math.pi*math.pi)*(sinc(s1) \
                                +1/(s1*s1)*(np.cos(s1)-1))
    # y[I1]=Omega*Omega/(4*math.pi*math.pi)*(sinc(s1)-1/2* (sinc(s1/2))**2)-y[I1]
    return y
```

[5]:
```
if (fbp_interactive):
    # Omega=1
    s=np.linspace(-10,10,256)
    y=ram_lak(s)
    plt.plot(s,y)
```



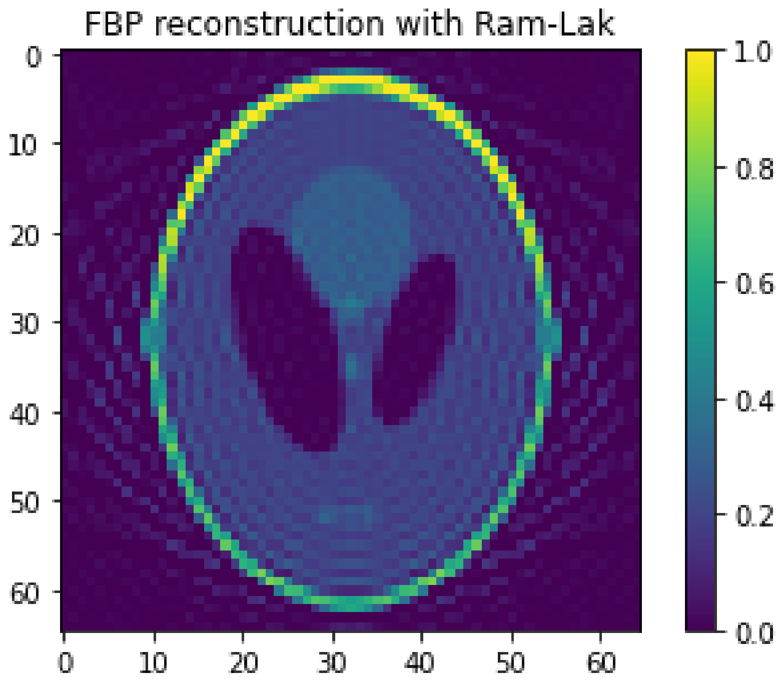
FBP implementation according to 6.15. Note that the resolution of the image is taken as the resolution of the canvas.

[6]:
```
def fbp(thetavec,q,R,filter_):
    img=np.zeros(canvas_X.shape)
    S=np.arange(-q,q+1)/q
    p=np.shape(thetavec)[0]
    for i in range(canvas_X.size):
        x1=canvas_X.flat[i]
        x2=canvas_Y.flat[i]
        sum=0
        for j in range(p):
            arg=x1*thetavec[j,0]+x2*thetavec[j,1]-S
            Z2=filter_(arg)
            sum=sum+np.sum(Z2*R[:,j])
        img.flat[i]=sum
        if (i % 1000 == 0):
            print(i,canvas_X.size)
    img=img*2*math.pi/(p*q)
    return img
```

[7]:
```
if (fbp_interactive):
    img=fbp(thetavec,q,R,ram_lak2)
```

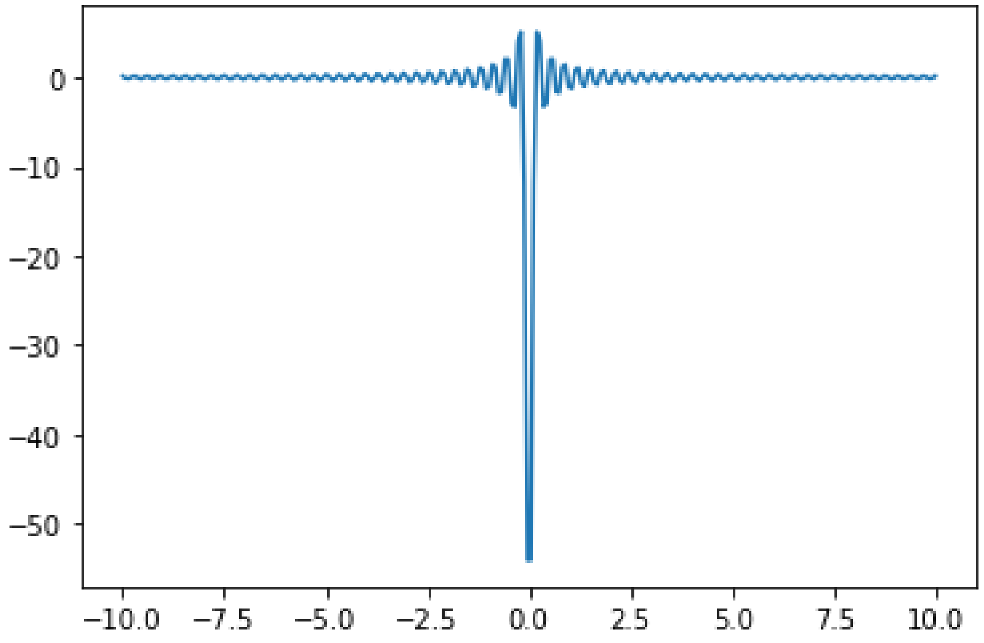
```
0 4225
1000 4225
2000 4225
3000 4225
4000 4225
```

[8]:
```
if (fbp_interactive):
    plt.imshow(img)
    plt.colorbar()
    plt.clim([0,1])
    plt.title('FBP reconstruction with Ram-Lak')
```



[9]:
```
def shepp_logan(s):
    s1=s*Omega
    I=np.where((abs(s-np.pi/2)<1e-6) | (abs(s+np.pi/2)<1e-6))
    I1=np.where((abs(s-np.pi/2)>=1e-6) & (abs(s+np.pi/2)>=1e-6))
    y=np.zeros(s.shape)
    y[I]=Omega**2/(2*math.pi**3)*(1/np.pi)
    s1 = s1[I1]
    y[I1]=Omega**2/(2*math.pi**3)*(np.pi/2-s1*np.sin(s1))/((np.pi/2)**2-s1**2)
    return y
```

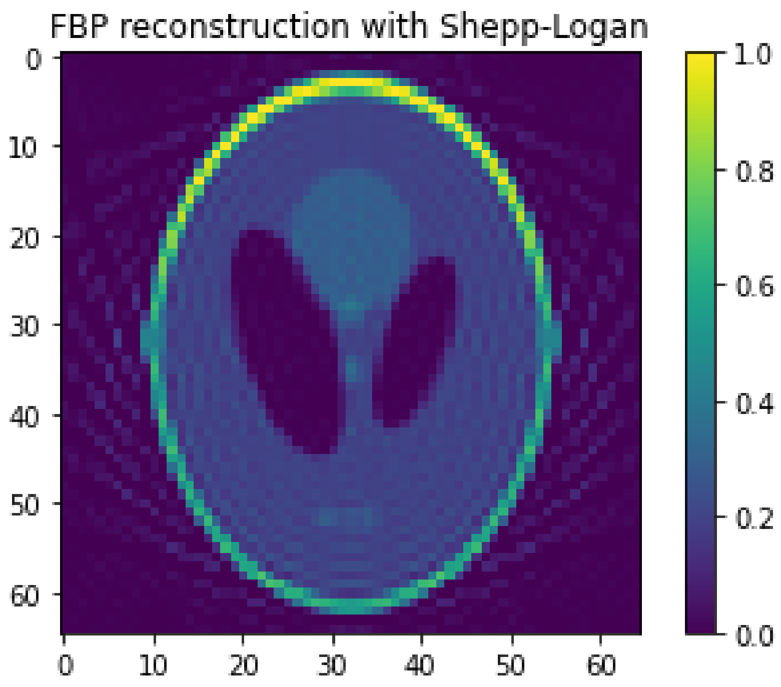
[10]:
```
if (fbp_interactive):
    # Omega=1
    s=np.linspace(-10,10,256)
    y=shepp_logan(s)
    plt.plot(s,y)
```



[11]:
```
if (fbp_interactive):
    img=fbp(thetavec,q,R,shepp_logan)
```

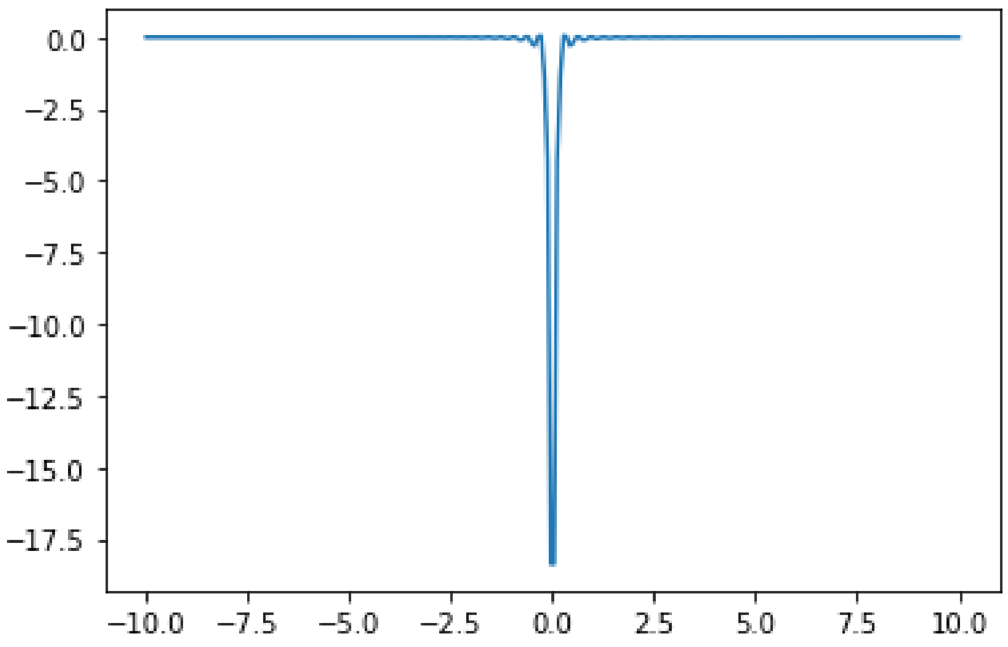
```
0 4225
1000 4225
2000 4225
3000 4225
4000 4225
```

[12]:
```
if (fbp_interactive):
    plt.imshow(img)
    plt.colorbar()
    plt.clim([0,1])
    plt.title('FBP reconstruction with Shepp-Logan')
```



```
[13]: def cosine(s):
          s1=s*Omega
          y=np.zeros(s.shape)
          y=Omega**2/(8*math.pi**2)*((sinc(s1+np.pi/2)+(np.cos(s1+np.pi/2)-1)/((s1+np.
       ↪pi/2)**2))+((sinc(s1-np.pi/2)+(np.cos(s1-np.pi/2)-1)/((s1-np.pi/2)**2))))
          return y
```

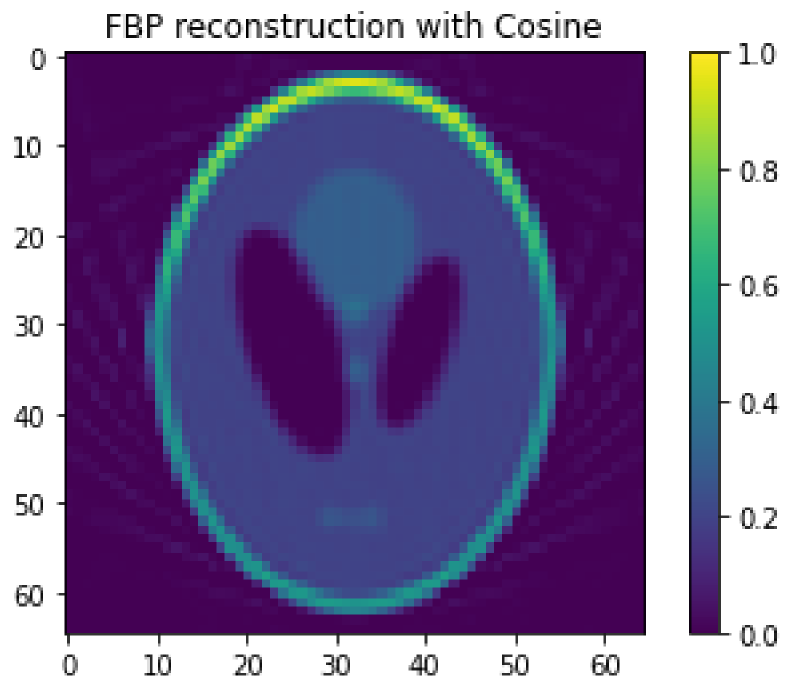
```
[14]: if (fbp_interactive):
          # Omega=1
          s=np.linspace(-10,10,256)
          y=cosine(s)
          plt.plot(s,y)
```



```
[15]: if (fbp_interactive):
          img=fbp(thetavec,q,R,cosine)
```

```
0 4225
1000 4225
2000 4225
3000 4225
4000 4225
```

```
[16]: if (fbp_interactive):
          plt.imshow(img)
          plt.colorbar()
          plt.clim([0,1])
          plt.title('FBP reconstruction with Cosine')
```


FBP reconstruction with Cosine
0
10
20
30
40
50
60
0
10
20
30
40
50
60
1.0
0.8
0.6
0.4
0.2
0.0

Shannon

January 31, 2021

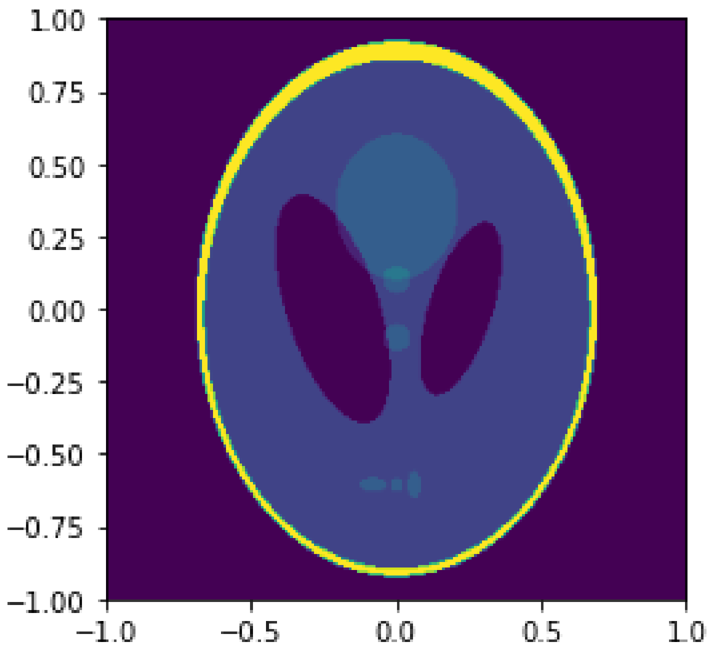
1 Shannon Sampling Theorem

1.1 Bandlimited Images

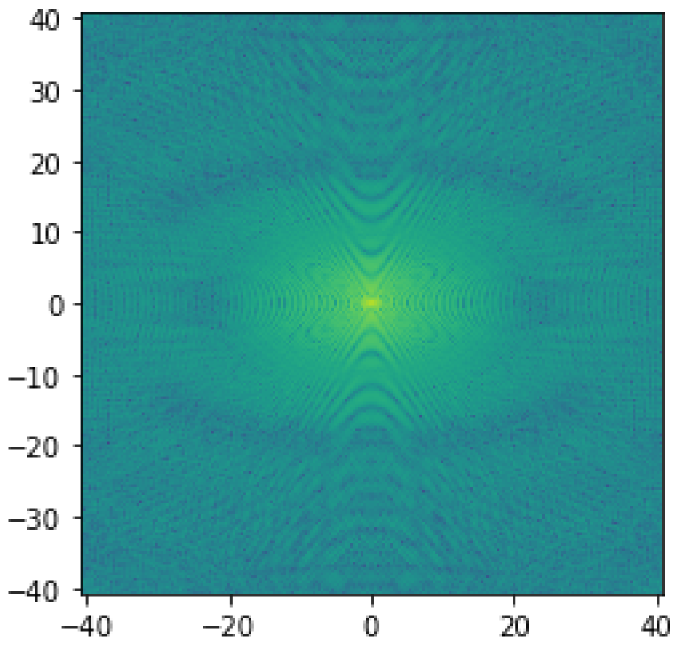
[46]:
```
radon_interactive=False
%run "../05 Radon Transform/radon.ipynb"
```

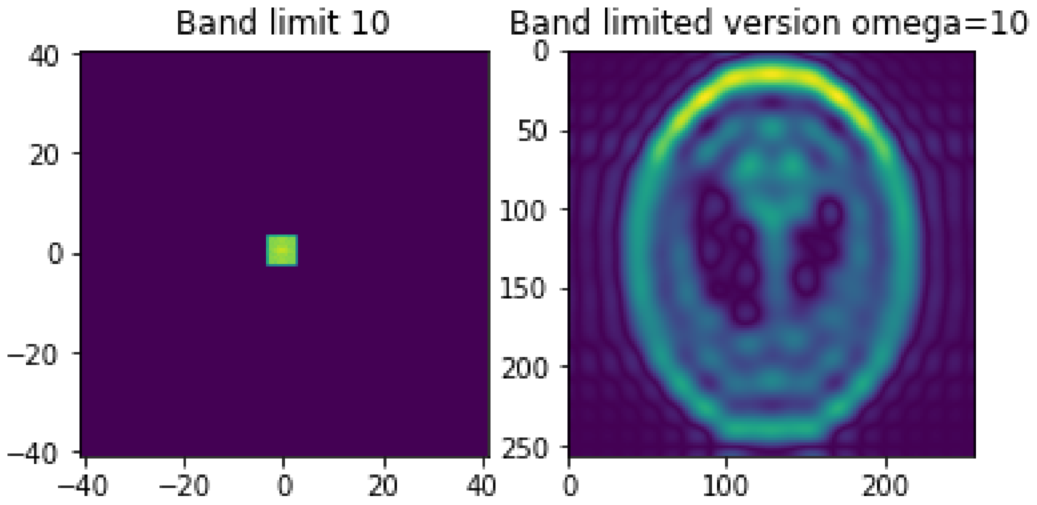
[47]:
```
N=128
x=np.arange(-N,N+1)
y=np.arange(-N,N+1)
Xgrid,Ygrid=np.meshgrid(x,y)
init_canvas(N,N)
draw_scene(ModifiedSheppLogan)
show_canvas()
```

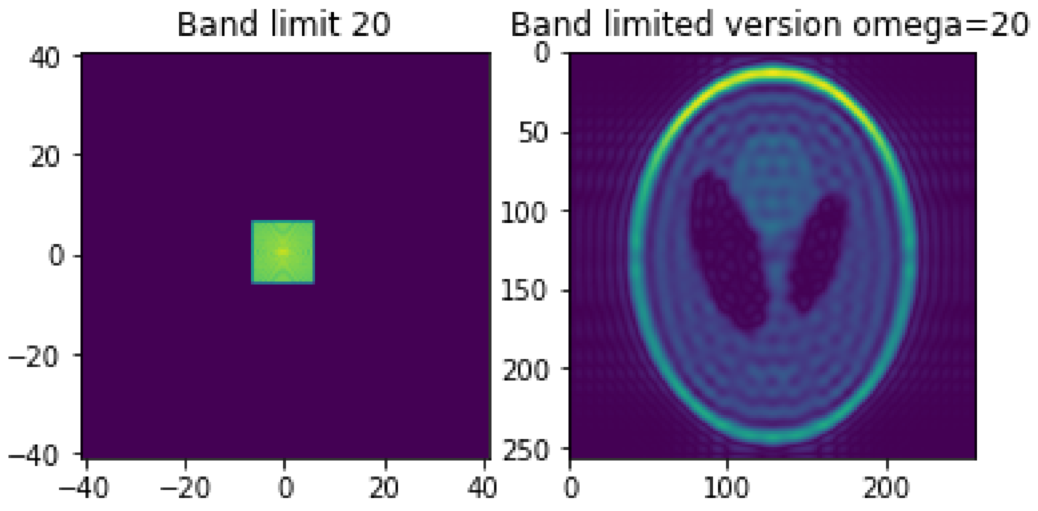
[47]:
```
<matplotlib.image.AxesImage at 0x7f1990cc3520>
```

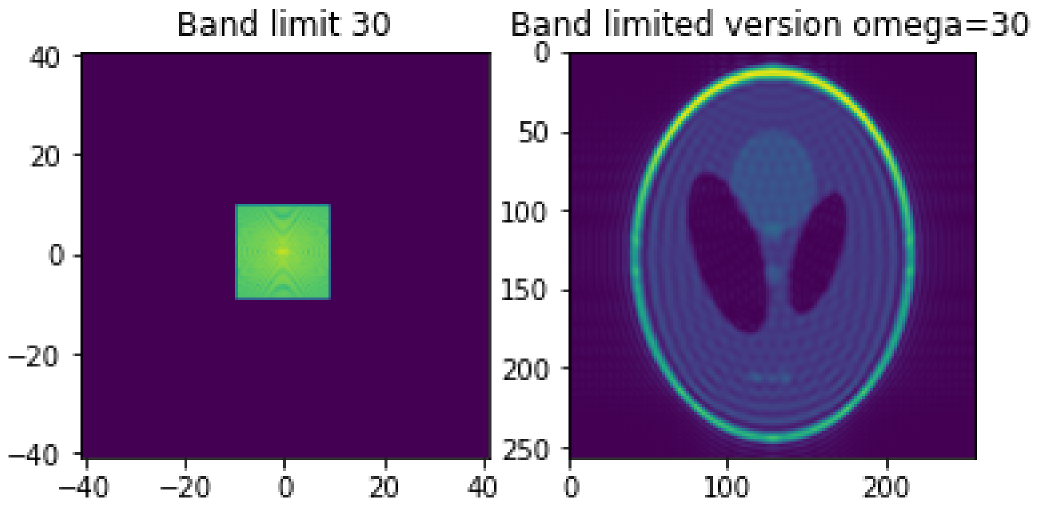


[48]:
```
def showfft(X):
    Y=np.fft.fftshift(X)
    extent=[-N/math.pi,N/math.pi,-N/math.pi,N/math.pi]
    plt.imshow(np.log(np.abs(Y)+1e-4),extent=extent)
    #plt.colorbar()

def bandlimit(X,omega):
    Y=np.fft.fft2(X)
    Y=np.fft.fftshift(Y)
    I=np.where((np.abs(Xgrid)<omega) & (np.abs(Ygrid)<omega))
    Z=np.zeros(X.shape,complex)
    Z[I]=Y[I]
    Z=np.fft.fftshift(Z)
    plt.subplot(121)
    showfft(Z)
    plt.title('Band limit '+str(omega))
    Z=np.fft.ifft2(Z)
    plt.subplot(122)
    plt.imshow(np.abs(Z))
    plt.title('Band limited version omega='+str(omega))
    plt.figure()
    return Z

F=np.fft.fft2(canvas)
showfft(F)
plt.figure()
bandlimit(canvas,10)
bandlimit(canvas,20)
bandlimit(canvas,30);
```


40
30
20
10
0
−10
−20
−30
−40
−40
−20
0
20
40


Band limit 10
Band limited version omega=10
40
20
0
−20
−40
−40
−20
0
20
40
0
50
100
150
200
250
0
100
200

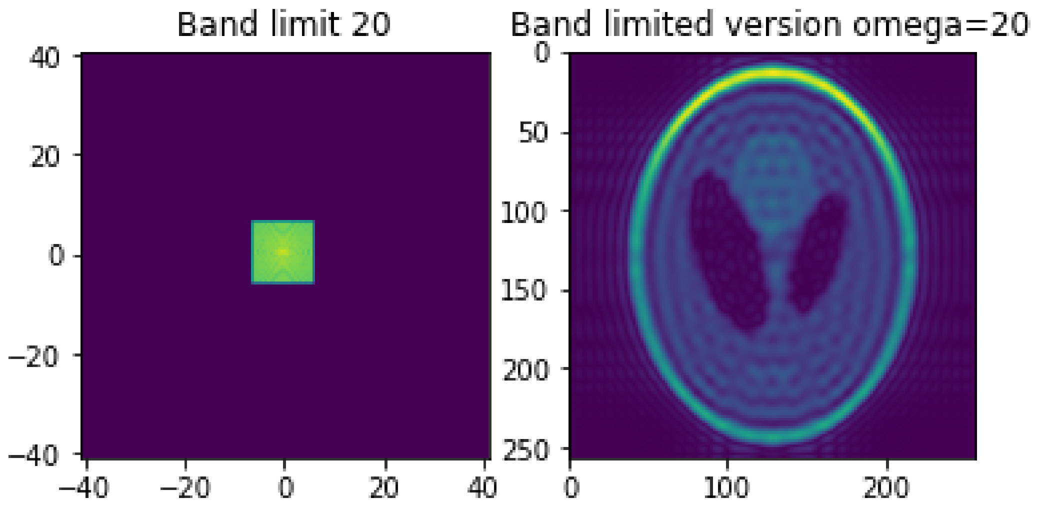




```
<Figure size 432x288 with 0 Axes>
```

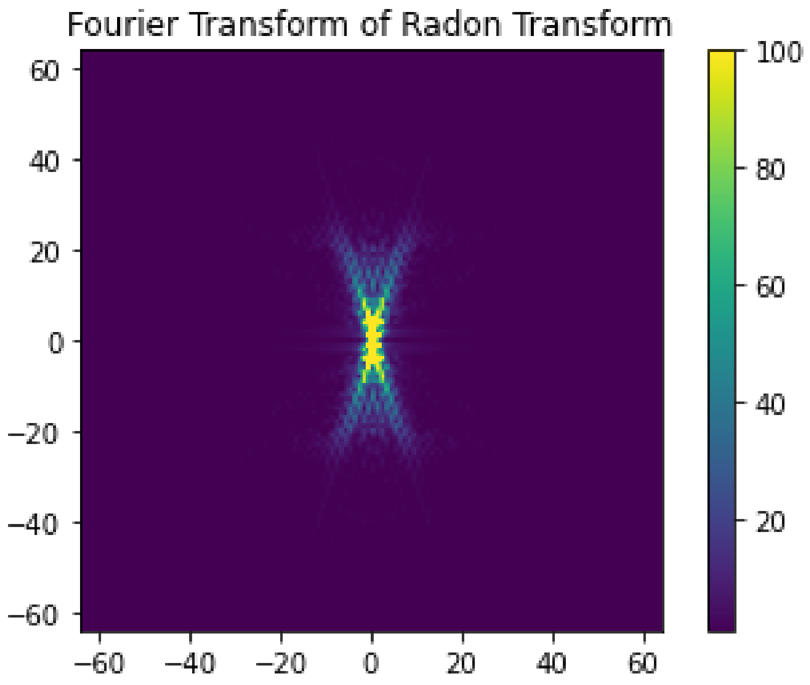
[49]:
```
slcanvas=canvas
```

[50]:
```
p=128
q=64
canvas=slcanvas
Z=bandlimit(canvas,20)
```



```
<Figure size 432x288 with 0 Axes>
```

[62]:
```
canvas=np.abs(Z)
R=parproj_canvas(p,q)
F=np.fft.fft2(R)
canvas=np.fft.fftshift((np.abs(F)))
A=show_canvas()
plt.colorbar()
plt.clim([1,100])
plt.title('Fourier Transform of Radon Transform')
A.set_extent([-q,q,-p/2,p/2])
```



Note that the “rings” in the image correspond to the Gibbs effect.

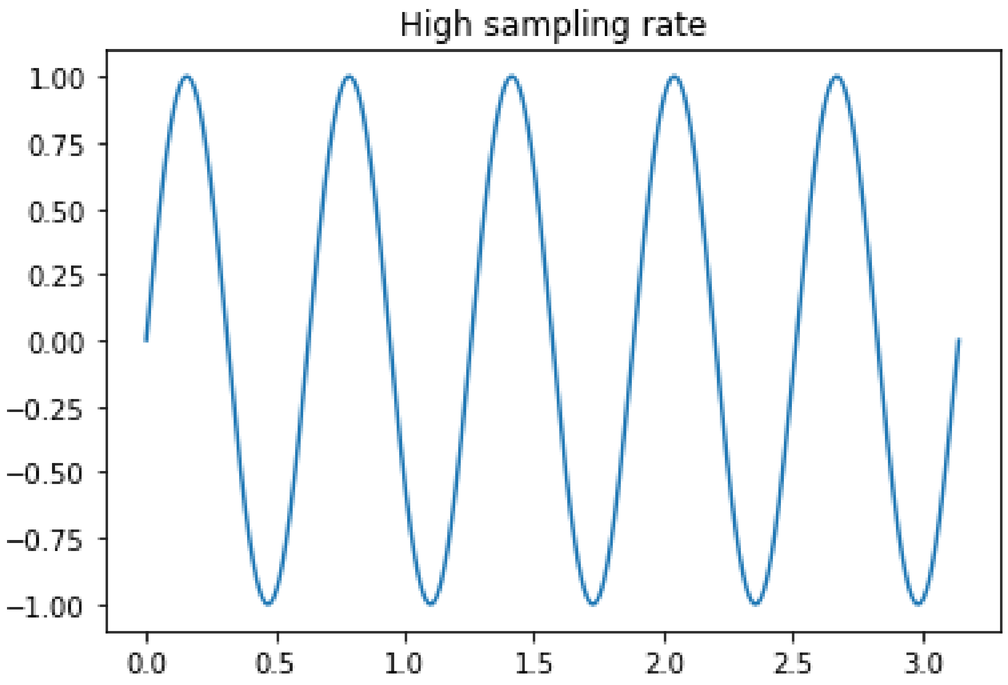
1.2 Sound

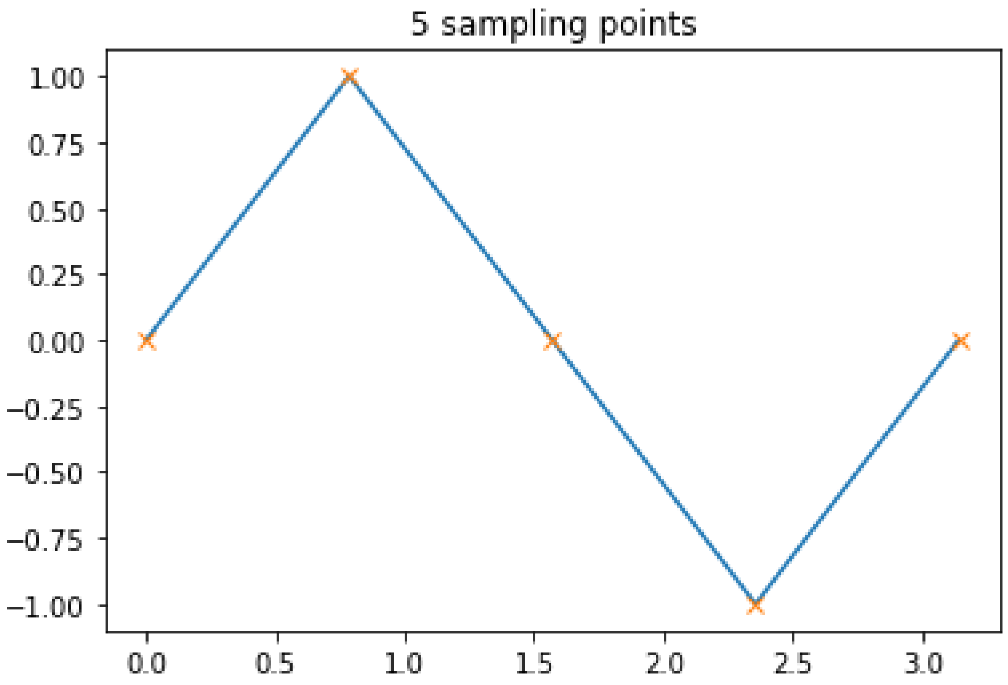
Acoustic signals must be sampled at twice the maximum frquency in the signal to represent them correctly. Example for the human ear: the upper frequency limit that an ear can hear is about 20 kHz. In a good recording device, first an analogue filter is used to cap frequencies beyond 22 kHz, leaving frequencies up to 20 kHz intact. So now the max frequency is 22 kHz, we need a sampling rate of 44 kHz to correctly represent our acoustic signal. This is the CD sampling rate.

It is important to note: It’s not important what you want to record, it is important what is in the signal (remember the error term in the Poisson summation formula!). If higher frequencies than what we can correctly represent exist, they falsify our result.

[4]:
```
N=10
x=np.linspace(0,math.pi,1024)
y=np.sin(N*x)
plt.plot(x,y)
plt.title('High sampling rate')
plt.figure()
x=np.linspace(0,math.pi,5)
y=np.sin(N*x)
plt.plot(x,y,x,y,'x')
plt.title('5 sampling points')
```

```
[4]: Text(0.5, 1.0, '5 sampling points')
```





```
[5]: import IPython.display as ipd

     def sinc(x):
         return np.sinc(x/math.pi)

     def resample(t,x,newrate,T):
         newt = np.linspace(0,T,int(T*newrate), endpoint=False)
         newx = np.zeros(newt.shape)
         h = t[1]-t[0]
         subt=np.array(range(0,len(t)))*math.pi
         for i in range(0,len(newt)):
             sum=np.sum(x*sinc(math.pi*newt[i]/h-subt))
             newx[i]=sum
             #for k in range(0,len(t)):
             #    sum=sum+x[k]*sinc(math.pi*newt[i]/h-k*math.pi)
             #if (i%1000==0):
             #    print(i)
         return newx
```

```
[6]:
     # Kammerton A
     sr = 22050 # sample rate
     sr= 800
     freq = 444
     T = 2.0    # seconds
     t = np.linspace(0, T, int(T*sr), endpoint=False) # time variable
     x = 0.8*np.sin(2*np.pi*freq*t)                # pure sine wave at 440 Hz
     newrate = 22050
     newx = resample(t,x,newrate,T)
     display(ipd.Audio(newx, rate=newrate)) # load a NumPy array
```

```
<IPython.lib.display.Audio object>
```

```
[7]: # Kammerton A
     sr = 22050 # sample rate
     sr= 1200
     freq = 444
     T = 2.0    # seconds
     t = np.linspace(0, T, int(T*sr), endpoint=False) # time variable
     x = 0.8*np.sin(2*np.pi*freq*t)+0.8*np.sin(2*np.pi*1.3*freq*t)                #␣
      ↪pure sine wave at 440 Hz
     newrate = 22050
     newx = resample(t,x,newrate,T)
     display(ipd.Audio(newx, rate=newrate)) # load a NumPy array
```

```
<IPython.lib.display.Audio object>
```

```
[ ]:
```

ART

January 31, 2021

1 ART - Algebraic Reconstruction Technique

We follow the description in the script and implement ART via the semidiscrete formulation rather than the discrete matrix formulation for performance reasons.

[1]:
```
bp_interactive=False
%run "../05 Radon Transform/backprojection.ipynb"
```

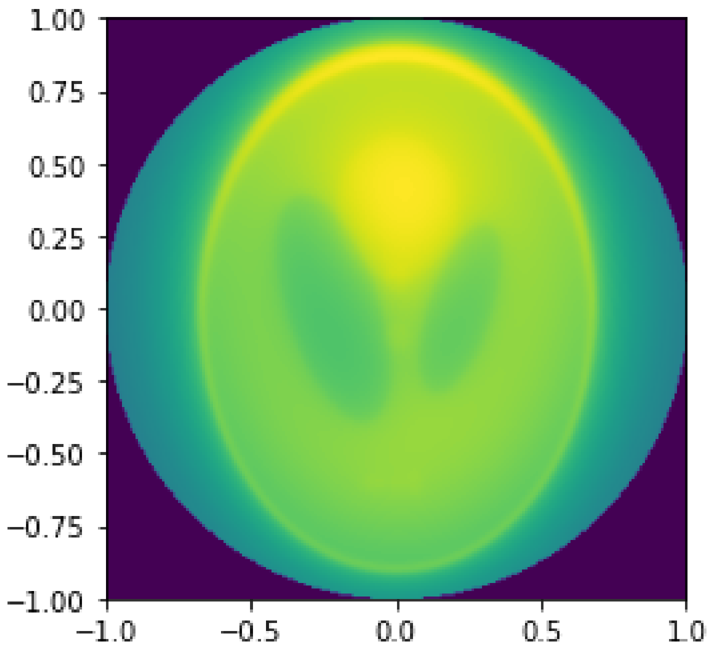
We generate the data. fak is the normalization factor in the definition of the ART backprojection step.

[2]:
```
phantom=ModifiedSheppLogan
q=64
p=int(q*math.pi)
phivec=np.arange(0,p)*math.pi/p
thetavec=np.array((np.cos(phivec),np.sin(phivec))).T
data=parproj(phantom,p=p,q=q)
X=np.linspace(-1,1,2*q+1)
fak=np.sqrt(1/np.array([1-X*X+1e-3])).T
```

For comparison: Unfiltered backprojection. This is the first Landweber step!

[3]:
```
N=128
init_canvas(N,N)
canvas=backproj(data,phivec=phivec)
show_canvas()
```

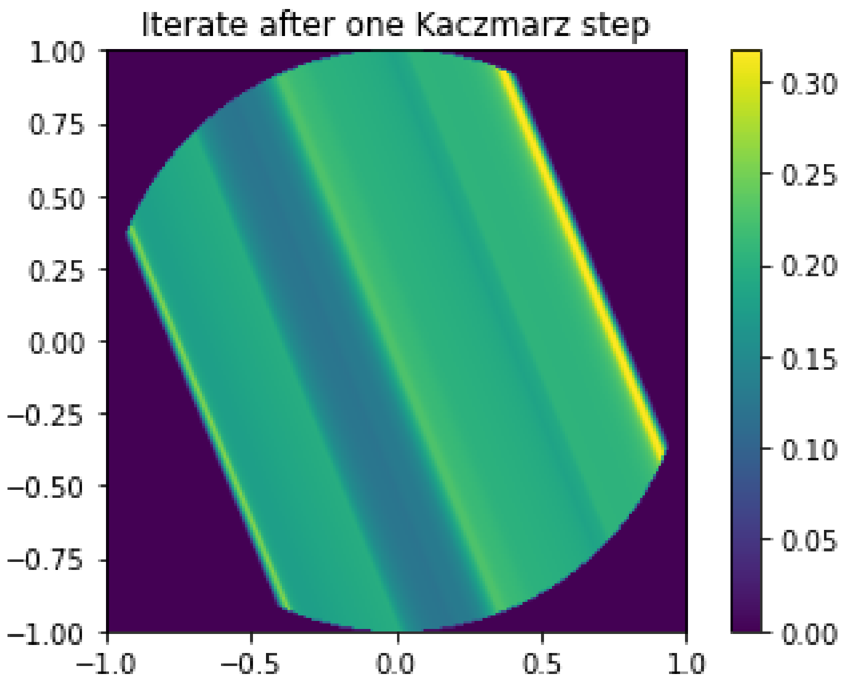
[3]:
```
<matplotlib.image.AxesImage at 0x7f6a9c933f10>
```

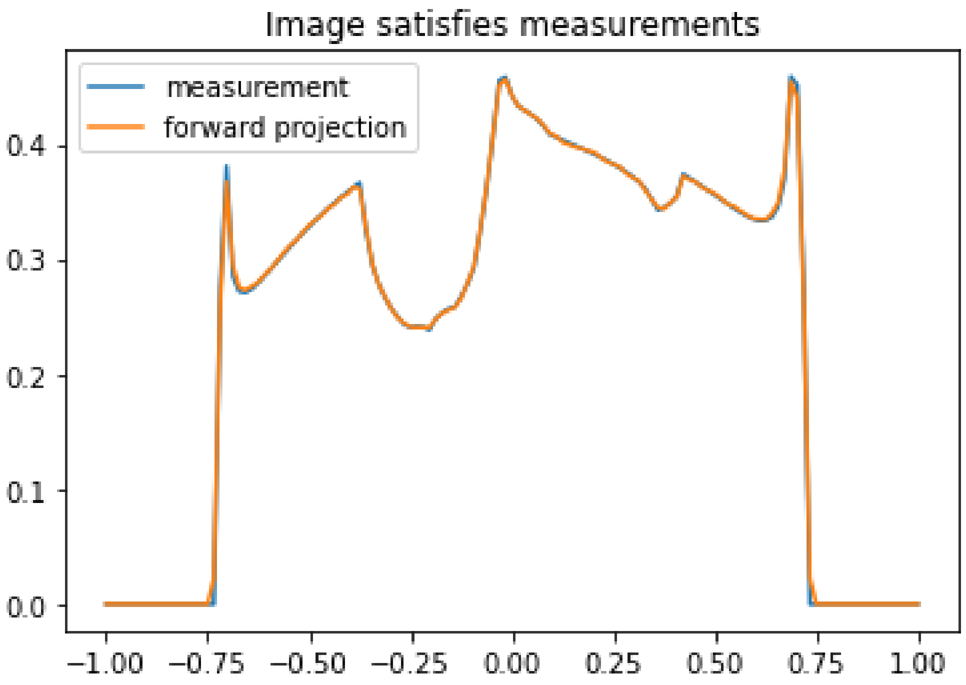


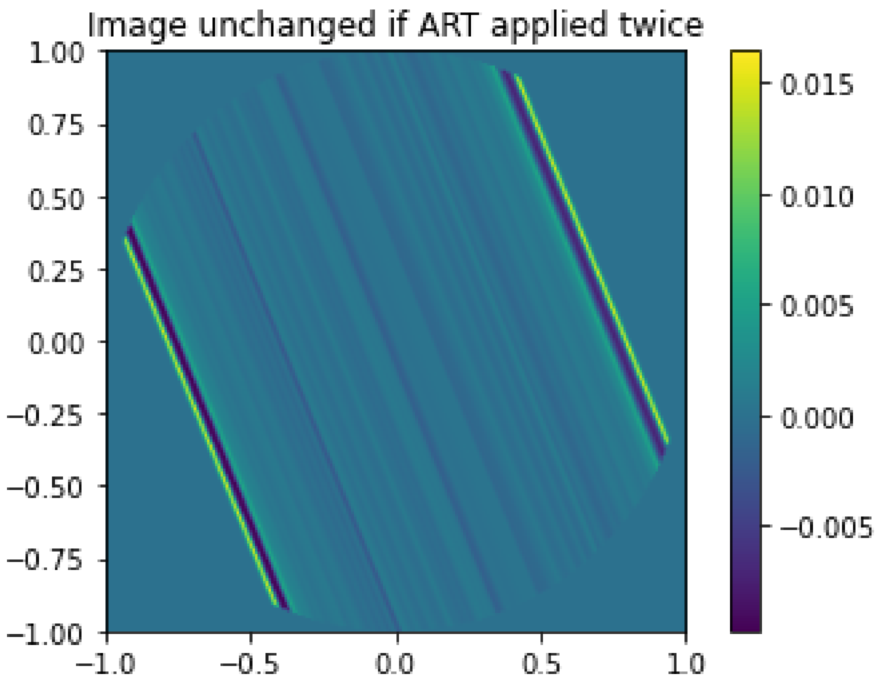
[4]:
```
# perform a single ART iteration
def ART_single_step(current,num):
    global data, phivec, p, q, canvas, N, X, omega
    # We use only one fixed source position
    single=data[:,num:num+1]
    angle=phivec[num:num+1]
    canvas=current
    g=single-parproj_canvas(p=1,q=q,phivec=angle)
    canvas=backproj(g/math.pi*fak,phivec=angle)
    current=current+omega*canvas
    return current
def ART_sweep(current,steps=None):
    global p
    if (steps is None):
        steps=p
    for num in range(0,steps):
        current=ART_single_step(current,num)
    return current

current=np.zeros([2*N+1,2*N+1])
num=25
omega=1
current=ART_single_step(current,num)
canvas=current
show_canvas()
```

```
plt.colorbar()
plt.title('Iterate after one Kaczmarz step')
plt.figure()
# Check that the discrete equation is satisfied
plt.title('Image satisfies measurements')
single=data[:,num:num+1]
angle=phivec[num:num+1]
g1=parproj_canvas(p=1,q=q,phivec=angle)
plt.plot(X,single,X,g1)
plt.legend(['measurement','forward projection'])
plt.figure()
# Check that image does not change when ART is applied twice for same data
current2=ART_single_step(current,num)
canvas=current-current2
show_canvas()
plt.colorbar()
plt.title('Image unchanged if ART applied twice')
```

```
[4]: Text(0.5, 1.0, 'Image unchanged if ART applied twice')
```



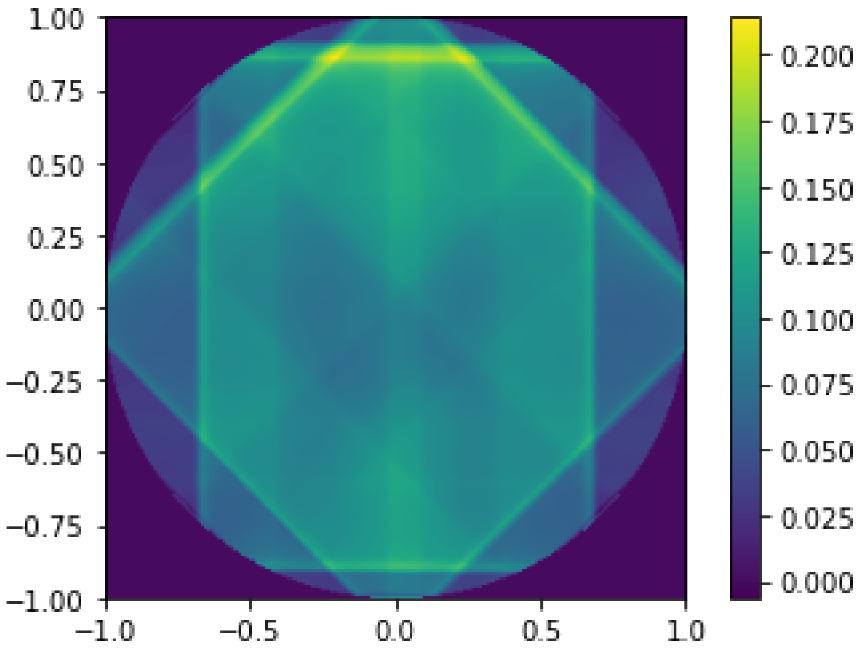




[13]:
```
omega=0.2
current=np.zeros([2*N+1,2*N+1])
```

```
#current=ART_sweep(current,steps=None)
#current=ART_sweep(current,steps=None)
#current=ART_sweep(current,steps=None)

current=ART_single_step(current,0)
#current=ART_single_step(current,1)
current=ART_single_step(current,p//2)
current=ART_single_step(current,p//4)
current=ART_single_step(current,3*p//4)
canvas=current
show_canvas()
#plt.clim([0,1])
plt.colorbar()
```

[13]:
```
<matplotlib.colorbar.Colorbar at 0x7f6a9c1cab50>
```

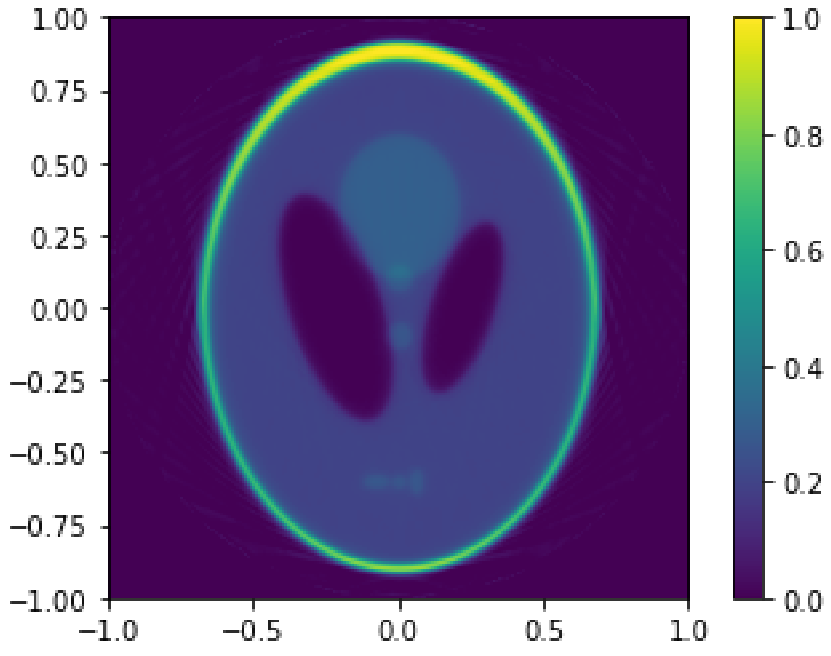


[14]:
```
data2=data.copy()
phivec2=phivec.copy()
```

[15]:
```
perms=np.random.permutation(p)
for i in range(0,p):
    P=perms[i]
    data[:,i]=data2[:,P]
    phivec[i]=phivec2[P]
```

[18]:
```
omega=0.2
current=np.zeros([2*N+1,2*N+1])
current=ART_sweep(current,steps=None)
#current=ART_sweep(current,steps=None)

canvas=current
P=show_canvas()
plt.clim([0,1])
plt.colorbar()
```

[18]:
```
<matplotlib.colorbar.Colorbar at 0x7f6a9c032820>
```



[ ]:
```
data=data2.copy()
phivec=phivec2.copy()
```

[ ]:
```
print(phivec2)
```

[ ]:
```

```

EM

January 31, 2021

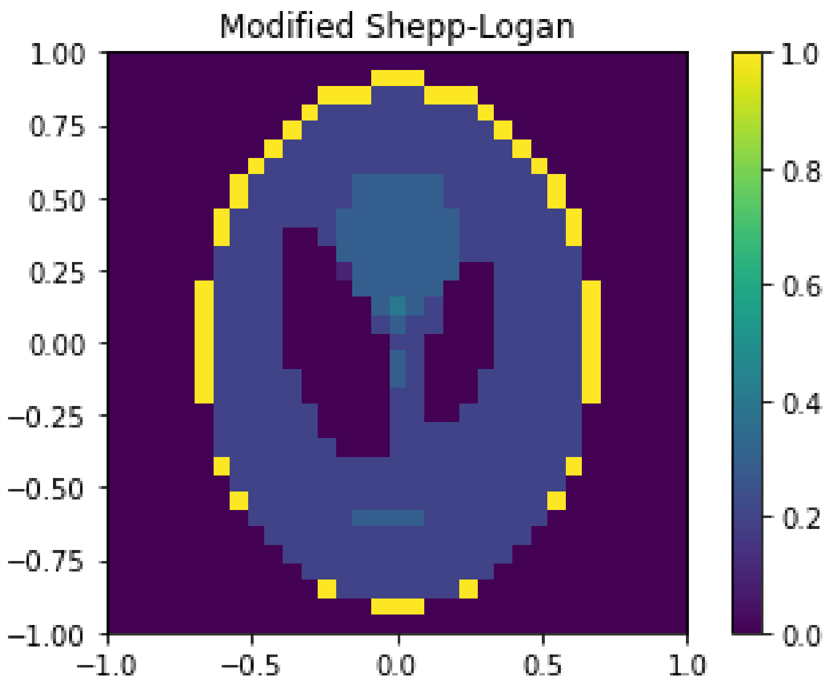
1 EM-Algorithm

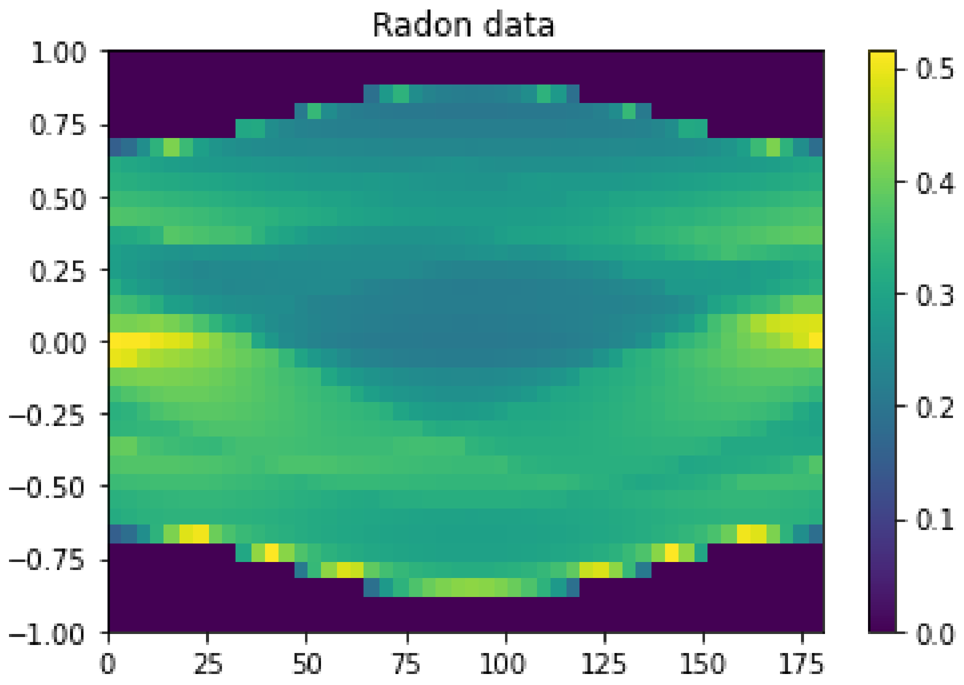
We compare the results of filtered backprojection and the EM algorithm.

[1]:
```
fbp_interactive=False
%run "../06 Sampling and Implementation/FBP.ipynb"
try: EM_interactive
except NameError: EM_interactive = True
```

Generate data as usual, see FBP. Note that Ω and so on should be globals.

[2]:
```
if (EM_interactive):
    N=16
    x=np.arange(-N,N+1)
    y=np.arange(-N,N+1)
    Xgrid,Ygrid=np.meshgrid(x,y)
    init_canvas(N,N)
    phantom=ModifiedSheppLogan
    draw_scene(phantom)
    show_canvas()
    original=canvas
    plt.colorbar()
    radonextent=[0,180,-1,1]
    plt.title('Modified Shepp-Logan')
    q=N
    p=int(np.floor(math.pi*q))
    Omega=p
    phivec=np.arange(0,p)*math.pi/p
    thetavec=np.array((np.cos(phivec),np.sin(phivec))).T
    R=parproj(phantom,p=p,q=q)
    R=parproj(phantom,q=q,phivec=phivec)
    plt.figure()
    plt.imshow(R,extent=radonextent,aspect='auto')
    plt.colorbar()
    plt.title('Radon data')
```

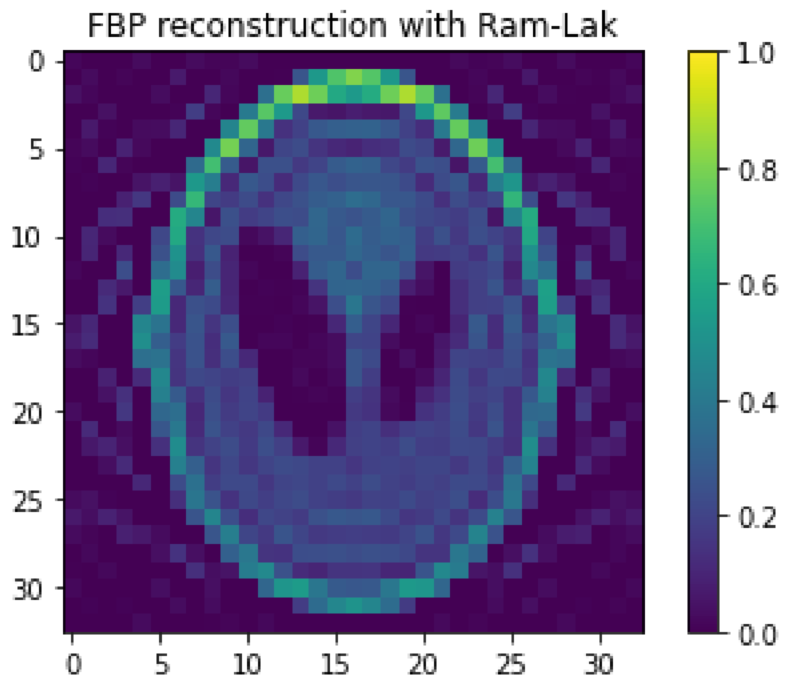




Do the filtered backprojection.

```
[3]: if (EM_interactive):
         img=fbp(thetavec,q,R)
         plt.imshow(img)
         plt.colorbar()
         plt.clim([0,1])
         plt.title('FBP reconstruction with Ram-Lak')
```

```
0 1089
1000 1089
```

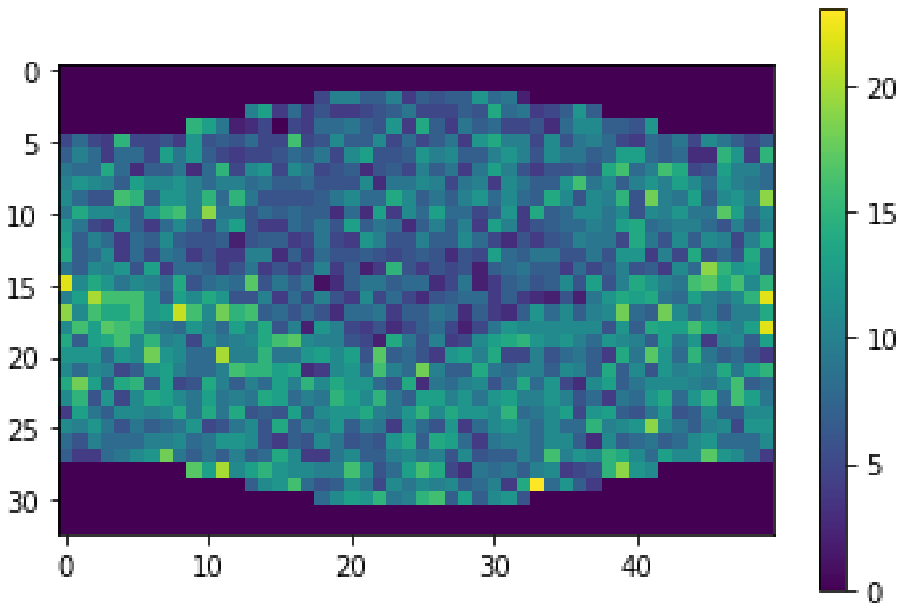


Simulate Poisson measurement, T0 seconds measurement time.

```
[22]: T0=30
      if (EM_interactive):
          Rpoisson=np.random.poisson(lam=T0*R)
```

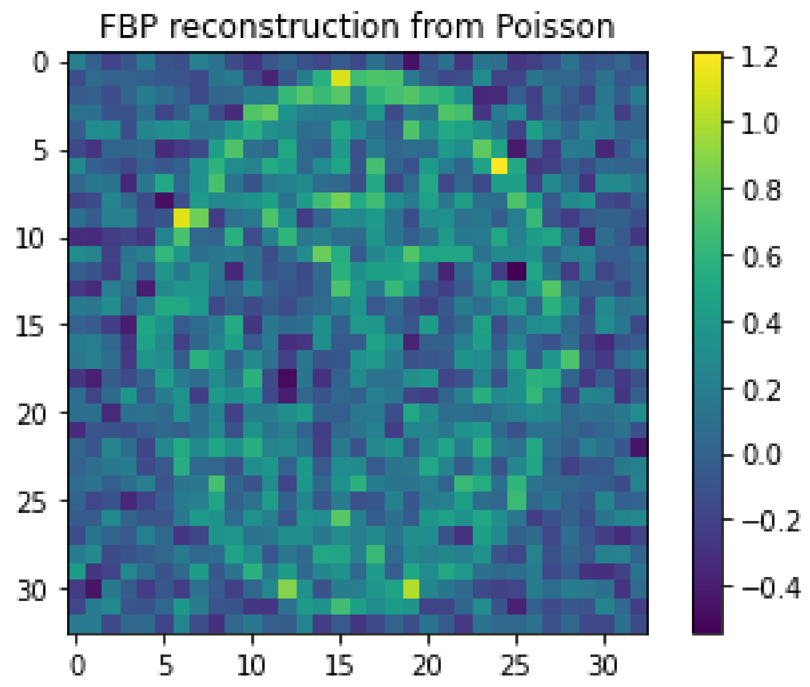
```
[23]: plt.imshow(Rpoisson)
      plt.colorbar()
```

```
[23]: <matplotlib.colorbar.Colorbar at 0x7fe515952070>
```



[24]:
```
if (EM_interactive):
    fbp_poisson=fbp(thetavec,q,Rpoisson)/T0
    plt.imshow(fbp_poisson)
    plt.colorbar()
    #plt.clim([0,1])
    plt.title('FBP reconstruction from Poisson')
```

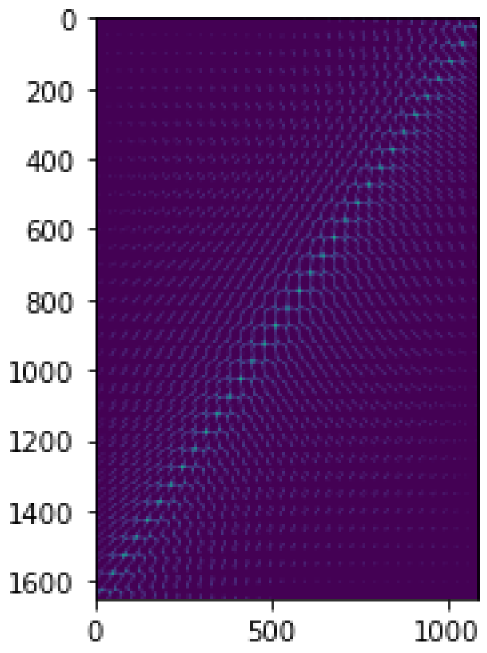
```
0 1089
1000 1089
```



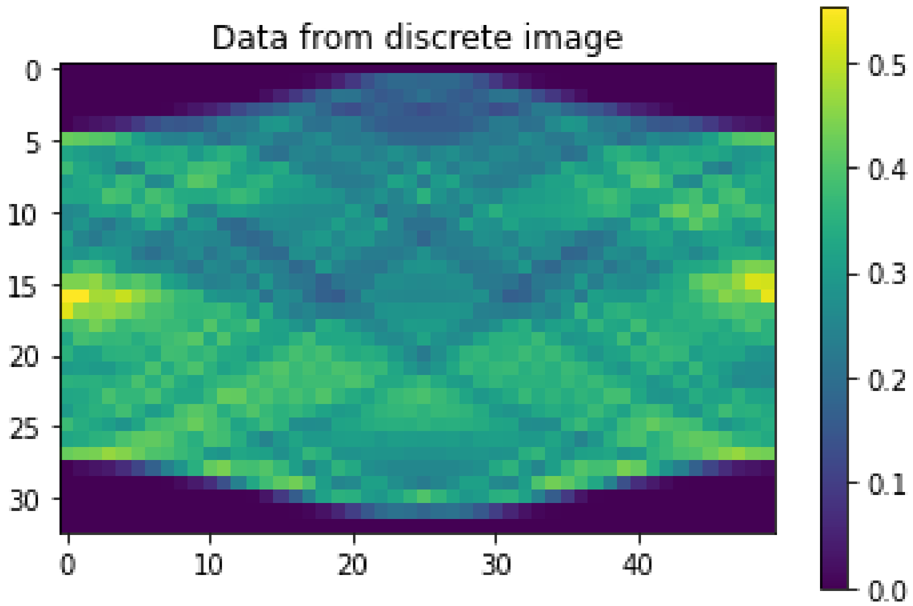
[25]:
```
# Generate Radon matrix.
def gen_matrix(p,q):
    init_canvas(canvas_N, canvas_M)
    N=canvas_X.size
    A=np.zeros([(2*q+1)*p,N])
    print(A.shape)
    for i in range(N):
        canvas.flat[i]=1
        R=parproj_canvas(p=p,q=q)
        if (i % 100 == 0):
            print(i)
        A[:,i]=R.flat
        canvas.flat[i]=0
    return A


if (EM_interactive):
    try: A
    except NameError: A=np.zeros(1)
    if (A.size!=(2*q+1)*p*canvas_X.size):
        A=gen_matrix(p,q)
```

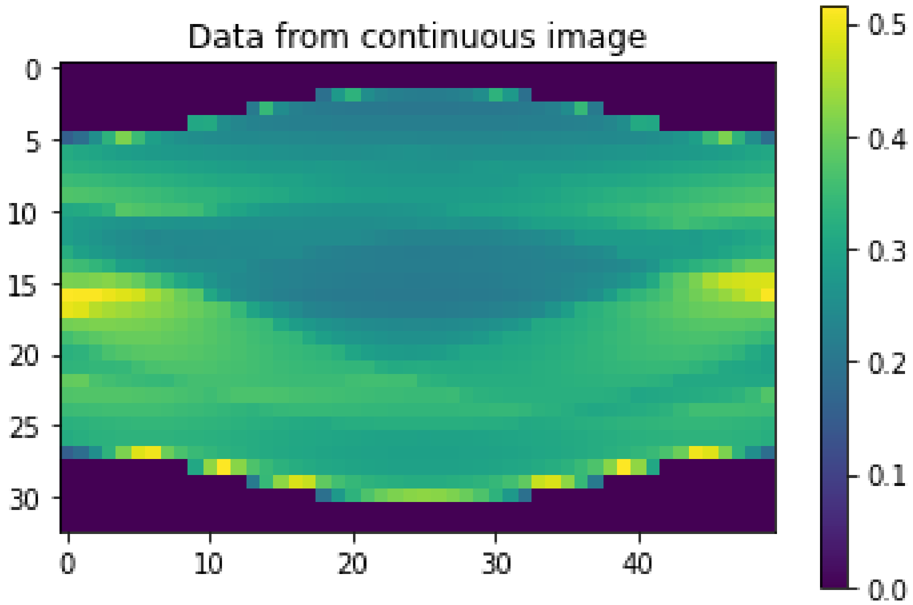
[26]:
```
plt.imshow(A)
```

```
[26]: <matplotlib.image.AxesImage at 0x7fe515b26b80>
```



```
[27]: if (EM_interactive):
          g=A.dot(original.flat)
          g=g.reshape(2*q+1,p)
          plt.imshow(g)
          plt.colorbar()
          plt.title('Data from discrete image')
          plt.figure()
          plt.imshow(R)
          plt.colorbar()
          plt.title('Data from continuous image')
```

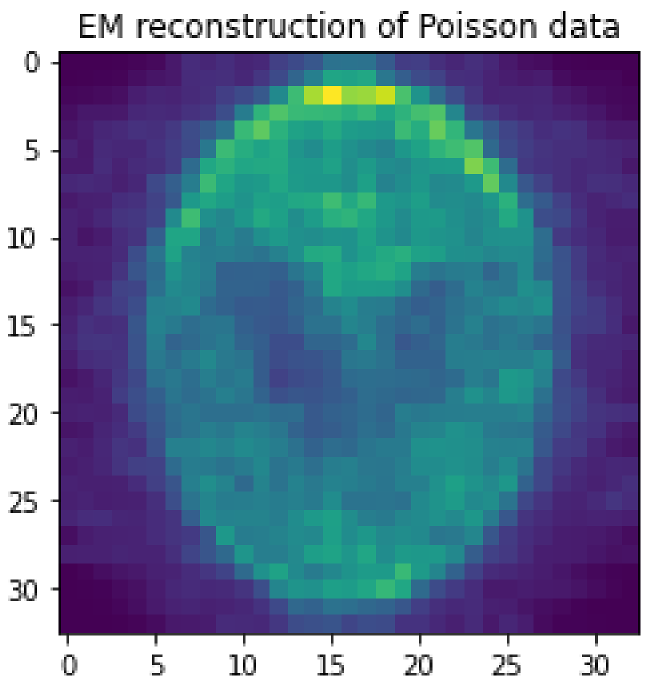




[28]:
```
g=np.ones((2*q+1)*p)
normalizer=1/A.T.dot(g)
#plt.imshow(normalizer.reshape(canvas_X.shape))
#plt.colorbar()
#plt.figure()
f=np.ones(canvas_X.shape)
```

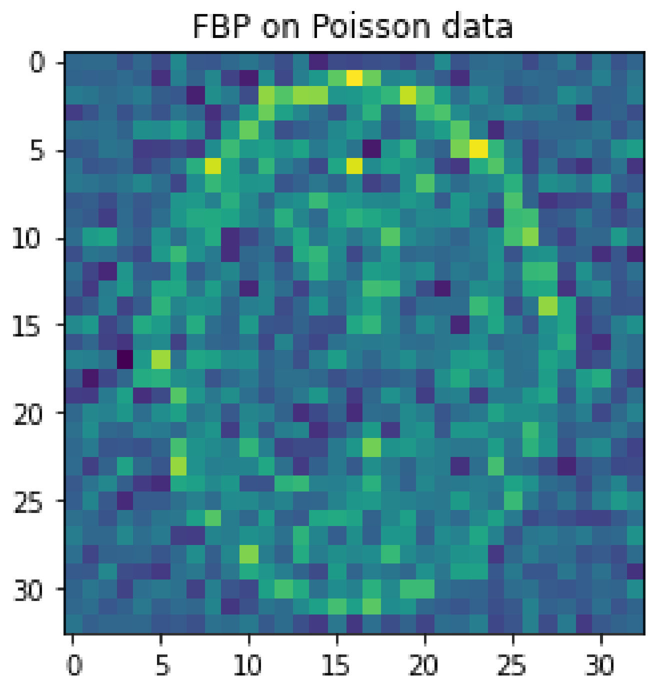
```
f=f.flatten()
data=Rpoisson.flatten()
omega=0.1
for i in range(0,30):
    f=f*(normalizer*A.T.dot(data/A.dot(f)))**omega
plt.imshow(f.reshape(canvas_X.shape))
plt.title('EM reconstruction of Poisson data')
```

```
[28]: Text(0.5, 1.0, 'EM reconstruction of Poisson data')
```



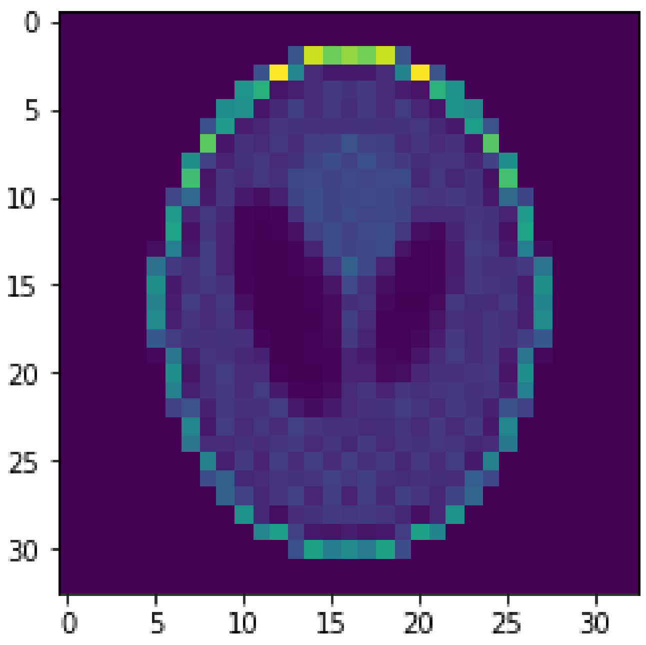
```
[14]: plt.imshow(fbp_poisson)
      plt.title('FBP on Poisson data')
```

```
[14]: Text(0.5, 1.0, 'FBP on Poisson data')
```



```
[12]: f=np.ones(canvas_X.shape)
      f=f.flatten()
      data=R.flatten()
      for i in range(0,1000):
          f=f*(normalizer*A.T.dot(data/A.dot(f)))**0.1
      plt.imshow(f.reshape(canvas_X.shape))
```

```
[12]: <matplotlib.image.AxesImage at 0x7fe517894310>
```


0
5
10
15
20
25
30
0
5
10
15
20
25
30

```
[ ]:
```

Random Images

January 31, 2021

1 Random Images

We investigate the role of the covariance matrix in statistical image models by generating random images. We look at normal distribution only.

[1]:
```
bp_interactive=False
%run "../05 Radon Transform/backprojection.ipynb"
```

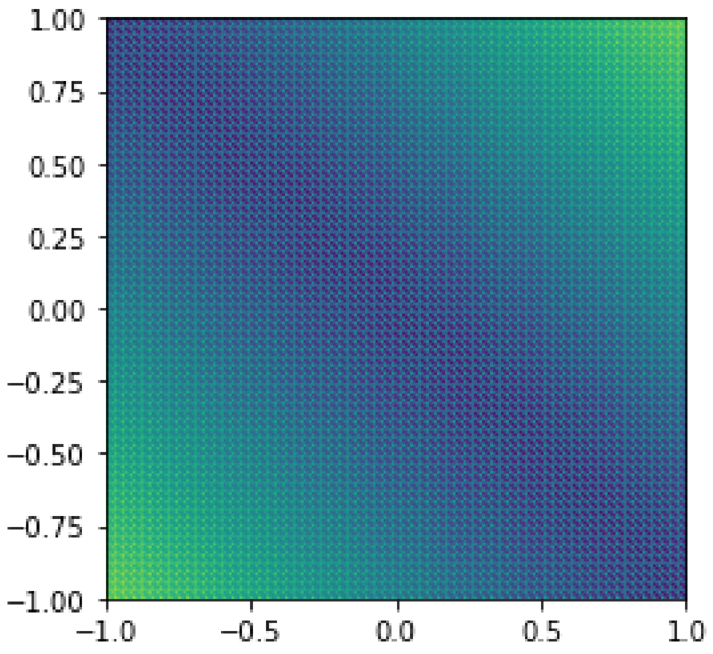
[2]:
```
N=32
init_canvas(N)
```

We interpret the image as a vector of pixel values and flatten the coordinates of each pixel. We setup a matrix where (i,k) contains the euclidean distance between Pixel i and Pixel k.

We should not set up this matrix (but compute it on-the-fly), but for simplicity and clarity, at this point we do.

[3]:
```
M=(2*N+1)*(2*N+1)
canvasX_flat=canvas_X.flatten()
canvasY_flat=canvas_Y.flatten()
D=np.zeros([M,M])
for i in range(M):
    #for k in range(M):
        #S[i,k]=1/(1+np.
 ↪hypot(canvasX_flat[i]-canvasX_flat[k],canvasY_flat[i]-canvasY_flat[k]))
    #S[i,:]=1/(1+np.
 ↪hypot(canvasX_flat[i]-canvasX_flat,canvasY_flat[i]-canvasY_flat))
    D[i,:]=np.hypot(canvasX_flat[i]-canvasX_flat,canvasY_flat[i]-canvasY_flat)
canvas=D
show_canvas()
```

[3]:
```
<matplotlib.image.AxesImage at 0x7fc13acdd8b0>
```



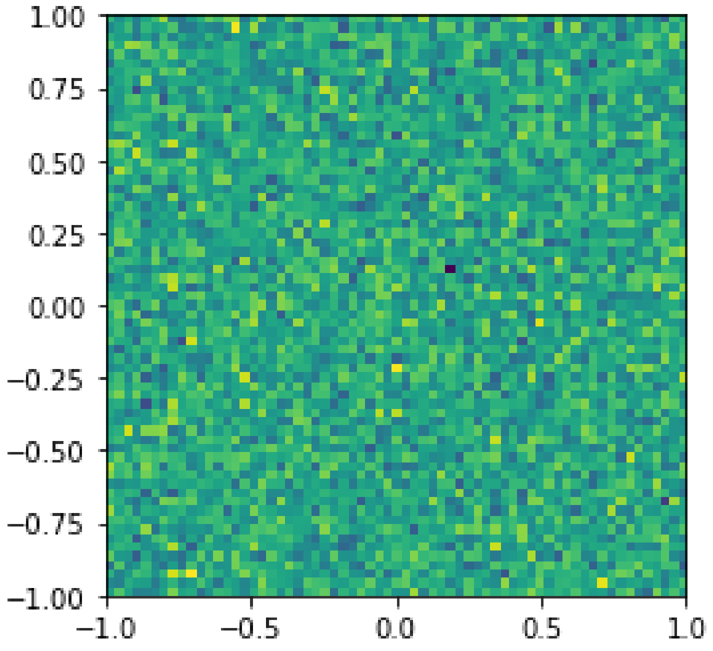
We construct a suitable covariance matrix. Neither is the model good, nor correct (in fact, we should draw the square root). But for demonstration purposes, it will be ok. Assume that the cov is S^2.

```
[4]: S=1/(1+D)
```

Generate a purely random image, normally distributed, cov=I.

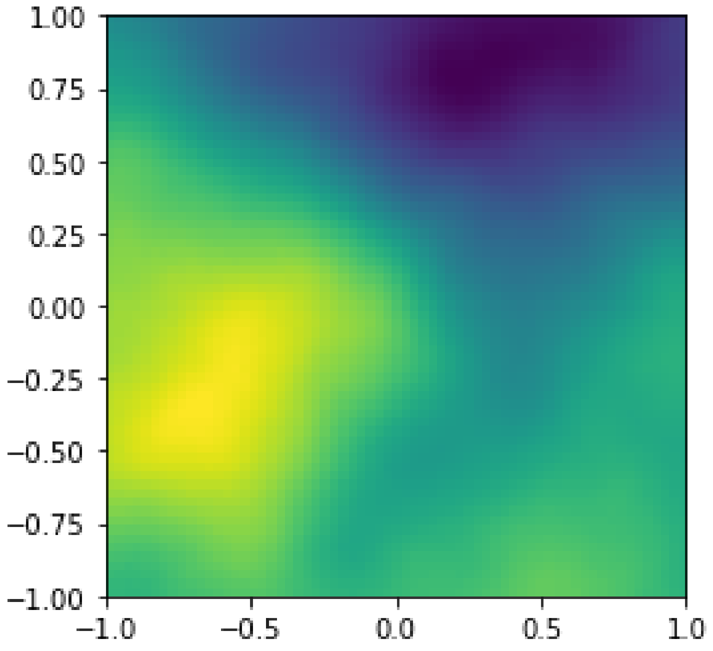
```
[7]: canvas=np.random.randn(2*N+1,2*N+1)
     canvas_flat=canvas.flatten()
     show_canvas()
```

```
[7]: <matplotlib.image.AxesImage at 0x7fc120279b20>
```



```
[8]: F=S.dot(canvas_flat)
     F=F.reshape([2*N+1,2*N+1])
     canvas=F
     show_canvas()
```

```
[8]: <matplotlib.image.AxesImage at 0x7fc1201cfc10>
```



Note: The matrix elements depend on $||x_i - x_k||$. So there is a simple analytical interpretation of this: Applying the matrix to the random noise is just a 2D-convolution.

```
[ ]:
```

Helmholtz

January 31, 2021

1 An initial value problem for the Helmholtz equation

We take a closer look at the problem

$$\Delta u + k^2 u = 0,\ u(x,0) = u_0(x), \frac{\partial u}{\partial y}(x,0) = 0$$

with k fixed and the initial value

$$u_0(x) = \sin(lx).$$

The analytical solution is given by

$$u(x,y) = \sin(lx)\cos(\sqrt{k^2 - l^2}y)$$

where the cosine is declared for complex numbers as usual and $\sqrt{-a^2} = i|a|$. Note that for this problem, we proved existence and stability of solutions.

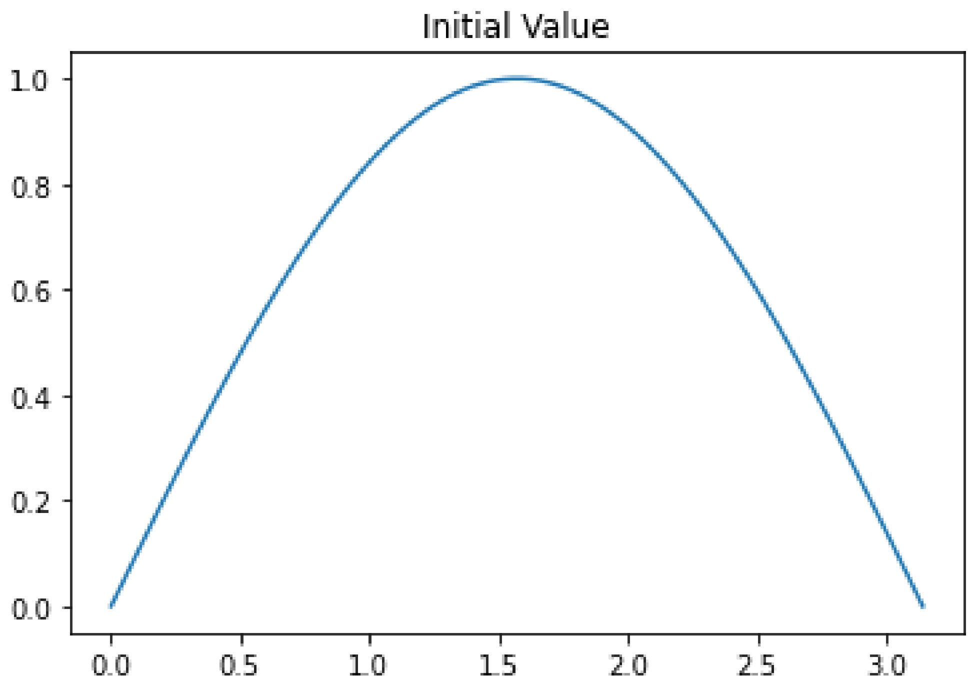
We plot this, note that the initial value is at the top and we use the colorbar depicted to the right of the 2D image.

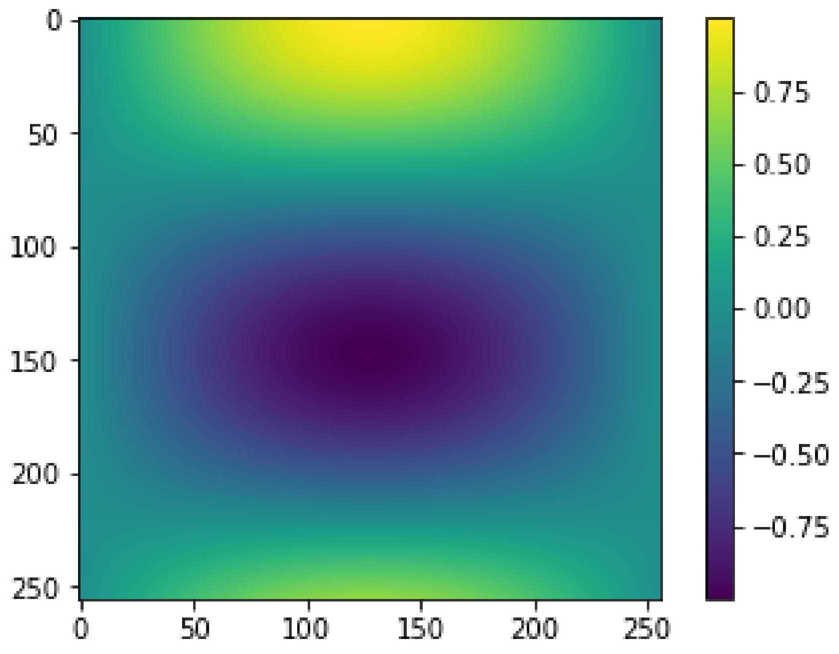
[150]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy
import math
import cmath
import scipy.fftpack
```

[154]:
```
k=2
l=1
N=256
X=np.linspace(0,math.pi,N)
Y=np.linspace(0,math.pi,N)
dx=math.pi/N
dy=dx

def u(x,y):
    return cmath.sin(l*x)*cmath.cos(cmath.sqrt(k*k-l*l)*y)

u0=[u(x,0) for x in X]
u0=np.array(u0)
```

```
plt.plot(X,np.real(u0))
plt.title('Initial Value')
plt.figure()
U=[[u(x,y) for x in X] for y in Y]
U=np.array(U)
plt.imshow(np.real(U))
plt.colorbar()
```

```
[154]: <matplotlib.colorbar.Colorbar at 0x7fc5d7e06690>
```

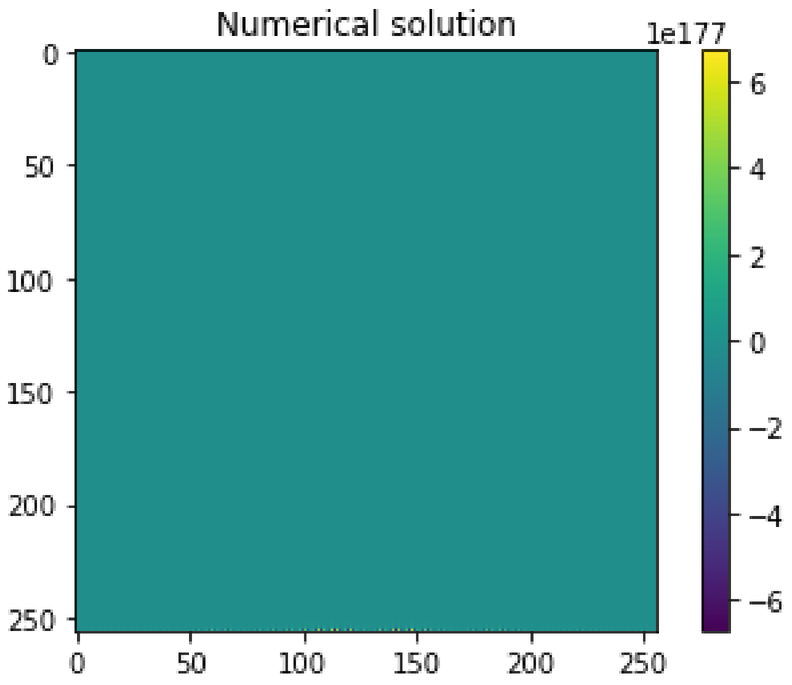


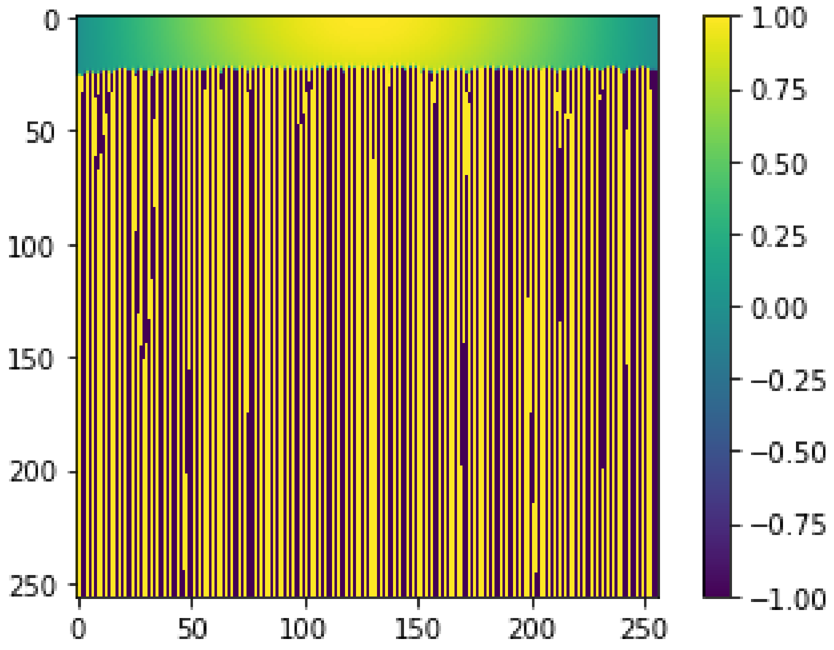


Now we use a standard 5 point star to compute the solution numerically.

[156]:
```
W=np.zeros(np.shape(U),complex)
W[0,:]=U[0,:]
W[1,:]=U[1,:]
dx2=dx*dx
k2=k*k
for i in range(1,N-1):
    for j in range(1,N-1):
        W[i+1,j]=-dx2*k2*W[i,j]-W[i-1,j]-W[i,j-1]-W[i,j+1]+4*W[i,j]
plt.imshow(np.real(W))
plt.colorbar()
plt.title('Numerical solution')
plt.figure()
plt.imshow(np.real(W),vmin=-1,vmax=1)
plt.colorbar()
```

[156]:
```
<matplotlib.colorbar.Colorbar at 0x7fc5d73eb7d0>
```





This is obviously instable. Absolute values in the solution are on the order of 10^{177}. We see that the approximation breaks down very early in the computation.

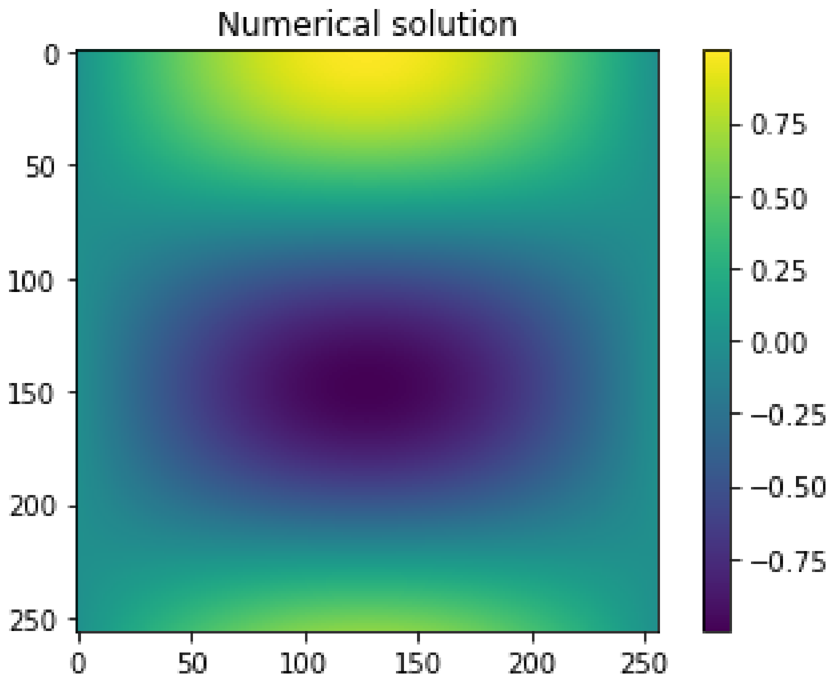
The reasons is simply that in the numerical code, we do not make use of the assumption that

Fourier coefficients of the solution with higher order vanish. By numerical errors, these are introduced into the numerical solution and quickly dominate the true solution.

The solution of course would be: In each marching step, make sure that the approximation is a sine series in x where the coefficients vanish for $j > l$. We repeat our computation and do exactly that.

```
[146]: W=np.zeros(np.shape(U),complex)
       W[0,:]=U[0,:]
       W[1,:]=U[1,:]
       dx2=dx*dx
       k2=k*k
       for i in range(1,N-1):
           for j in range(1,N-1):
               W[i+1,j]=-dx2*k2*W[i,j]-W[i-1,j]-W[i,j-1]-W[i,j+1]+4*W[i,j]
           T=W[i+1,:]
           S=scipy.fftpack.dst(T)
           for p in range(2,N):
               S[p]=0
           T=scipy.fftpack.idst(S)/(2*N)
           W[i+1,:]=T
       plt.imshow(np.real(W[:,:]))
       plt.colorbar()
       plt.title('Numerical solution')
       print(np.max(np.abs(W-U)))
```

```
0.016113037608403213
```



We notice that this time, everything goes well, the numerical approximation is in fact very good.

2 Conclusion

Introducing our a priori condition, we find that the solution can be computed in a stable way. However, we need to make use of the a priori assumption also in the numerical code by deleting coefficents in the series of higher ordern. This process is called filtering.

We note that the true solution without the a priori condition can be written as $u = u_1 + u_2$, where u_1 contains the coefficients of lower order and is stable, while u_2 contains the coefficients of higher order and cannot be computed in a stable way.

```
[ ]:
```

```
[ ]:
```

FTseries

January 31, 2021

1 FT series

Analytical Fourier Transform and Fourier series are closely related, so very similar computation rules hold.

Let $f \in L_2([-\pi, \pi]$. Let $\widehat{f}_k$ the kth Fourier coefficent of f.

1. Compute the Fourier coefficients of $f_a(x) := \chi_{[-a,a]}(x)$.
2. Let $f_b(x) = f(x+b)$. Compute $\widehat{f}_{bk}$ in terms of $\widehat{f}_k$.
3. Let $u(x) := f_a(x) - f_a(x-\pi)$ (periodically continued on $[-\pi, \pi]$). Compute the Fourier coefficients of u and prove that

$$u(x) = \sum_{k=0}^{\infty} c_k u_k(x),\ u_k(x) = \frac{2}{\sqrt{\pi}} \cos((2k+1)x), v_k(x) = \frac{2}{\sqrt{\pi}} \sin((2k+1)x), \sigma_k = \frac{1}{2k+1}.$$

Note: (u_k, v_k, σ_k) is the singular system for the antiderivative we computed in the example for theorem 2.33 (for simplicity here on the interval $[0, \pi/2]$).

Note for the next exercise: If you cannot compute it:

$$c_k = \frac{2a}{\sqrt{\pi}} sinc((2k+1)a).$$

[1]:
```
import numpy as np
import sympy
import math
import matplotlib.pyplot as plt
import cmath
```

Find the correct scaling factor for 3.

[2]:
```
x=sympy.symbols('x')
print(sympy.integrate(sympy.cos(x)*sympy.cos(x),(x,0,sympy.pi/2)))
```

```
pi/4
```

Kann man auch schnell von Hand machen, aber sympy macht es auch. $\widehat{f}_k = \frac{a}{\pi} sinc(ak)$.

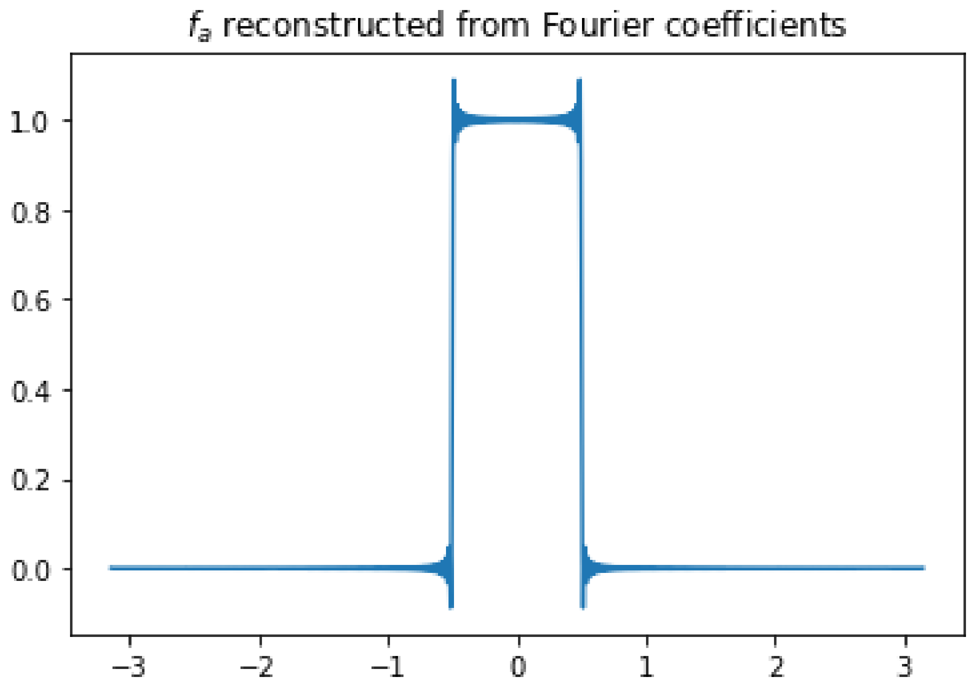
[3]:
```
k=sympy.symbols('k',positive=True)
x=sympy.symbols('x')
```

```
a=sympy.symbols('a')
print(sympy.simplify(1/(2*sympy.pi)*sympy.integrate(sympy.exp(sympy.
↪I*k*x),(x,-a,a))))
```

```
sin(a*k)/(pi*k)
```

Kurzer Check. Achtung, python hat die bekiffte Ingenieurs-Definition von sinc. Wir nehmen nicht die FFT.

```
[4]: a=0.5
N=1024
M=256
X=np.linspace(-math.pi,math.pi,N,endpoint=True)
koeff=np.zeros(2*M+1)
for k in range(-M,M+1):
    koeff[k+M]=np.sinc(a*k/math.pi)*a/math.pi

Y=np.zeros(X.shape,'complex')
for i in range(0,N):
    x=X[i]
    summe=0
    for k in range(-M,M+1):
        summe=summe+koeff[k+M]*cmath.exp(1j*k*x)
    Y[i]=summe
plt.plot(X,np.real(Y))
plt.title('$f_a$ reconstructed from Fourier coefficients');
```



Teil 2:

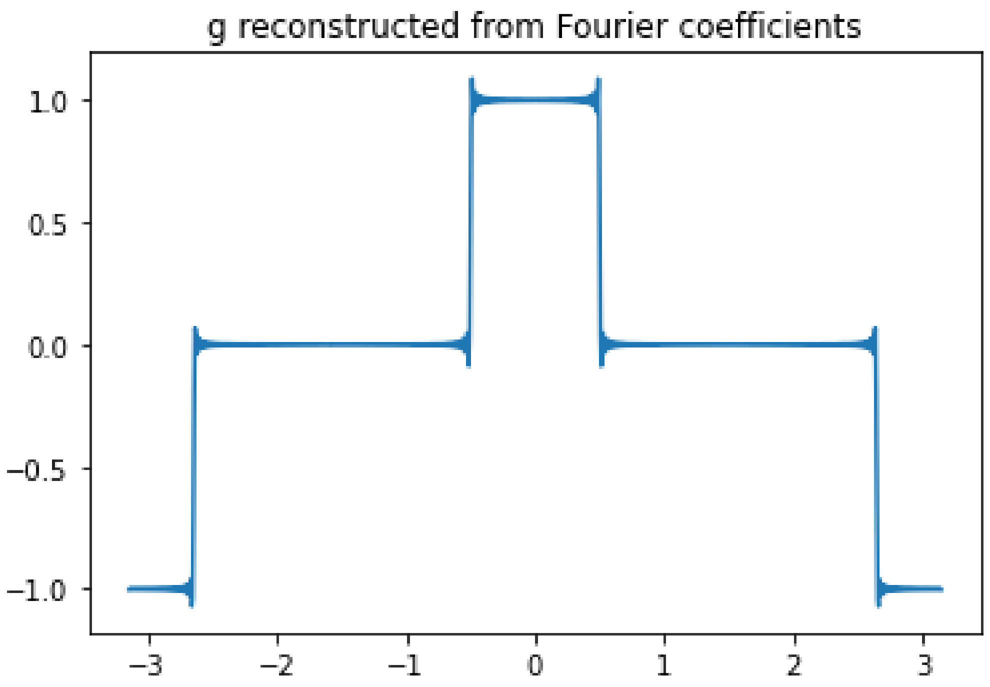
$$\widehat{f_b}_k = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x+b)e^{-ikx}\,dx = e^{ikb}\widehat{f}_k.$$

Teil 3:

$$\widehat{g}_k = \widehat{f_a}_k - (-1)^k\widehat{f_a}_k = \widehat{f_a}_k(1-(-1)^k.$$

[5]:
```python
a=0.5
N=1024
M=256
X=np.linspace(-math.pi,math.pi,N,endpoint=True)
koeff=np.zeros(2*M+1)
for k in range(-M,M+1):
    koeff[k+M]=np.sinc(a*k/math.pi)*a/math.pi*(1-(-1)**k)

Y=np.zeros(X.shape,'complex')
for i in range(0,N):
    x=X[i]
    summe=0
    for k in range(-M,M+1):
        summe=summe+koeff[k+M]*cmath.exp(1j*k*x)
    Y[i]=summe
plt.plot(X,np.real(Y))
plt.title('g reconstructed from Fourier coefficients')
```

[5]:
```
Text(0.5, 1.0, 'g reconstructed from Fourier coefficients')
```

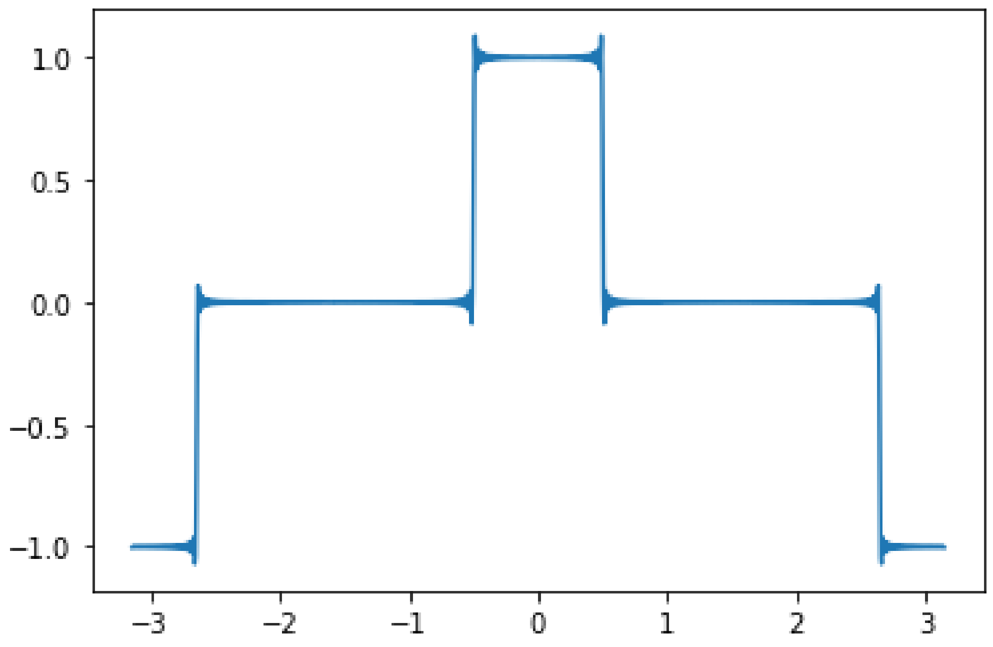


Offensichtlich gilt $\widehat{g}_{2k} = 0$ und $\widehat{g}_{2k+1} = 2\widehat{f}_{k+1}$. Aufgrund der Eigenschaften des sinc gilt zusätzlich $f_k = f_{-k}$. Also hat g die Darstellung

$$g = \sum_k 2 f_{2k+1} \cos((2k+1)\pi).$$

[6]:
```
a=0.5
N=1024
M=256
X=np.linspace(-math.pi,math.pi,N,endpoint=True)
koeff=np.zeros(M)
for k in range(0,M):
    koeff[k]=np.sinc(a*(2*k+1)/math.pi)*a*4/math.pi

Y=np.zeros(X.shape,'complex')
for i in range(0,N):
    x=X[i]
    summe=0
    for k in range(0,M//2):
        summe=summe+koeff[k]*math.cos((2*k+1)*x)
    Y[i]=summe
plt.plot(X,np.real(Y))
```

[6]:
```
[<matplotlib.lines.Line2D at 0x7f5b04412250>]
```



Write a program that, given the singular system (σ_k, u_k, v_k), $k = 0 \dots M$, of a compact operator K on $[0, \pi/2]$ and the coefficients $c'_k = (g, v_k)$, computes the minimum norm solution K^+g, Truncated SVD, Tikhonov regularized solution etc.

Test your program by implementing the singular system from exercise 3. Set $M = 128$ and take as test function the function u from exercise 3. Set $g = Ku$ and compute its coefficients c'_k from c_k using the singular system. Plot u and g on a grid on $[0, \pi/2]$ and make sure that this is correct.

Set $gn = g + n$ where n is random normal noise with standard deviation 1e-3. Compute the minimum norm solution and the regularized solutions. Try to find values of the regularization parameter that give good results. Plot these on a grid, and provide additional plots for values too big or small.

[7]:
```
N=1024
M=128
a=0.5

def uk(k,x):
    return math.cos((2*k+1)*x)*2/math.sqrt(math.pi)

def vk(k,x):
    return math.sin((2*k+1)*x)*2/math.sqrt(math.pi)

sigma=np.zeros(M)
koeff=np.zeros(M)
gkoeff=np.zeros(M)
```

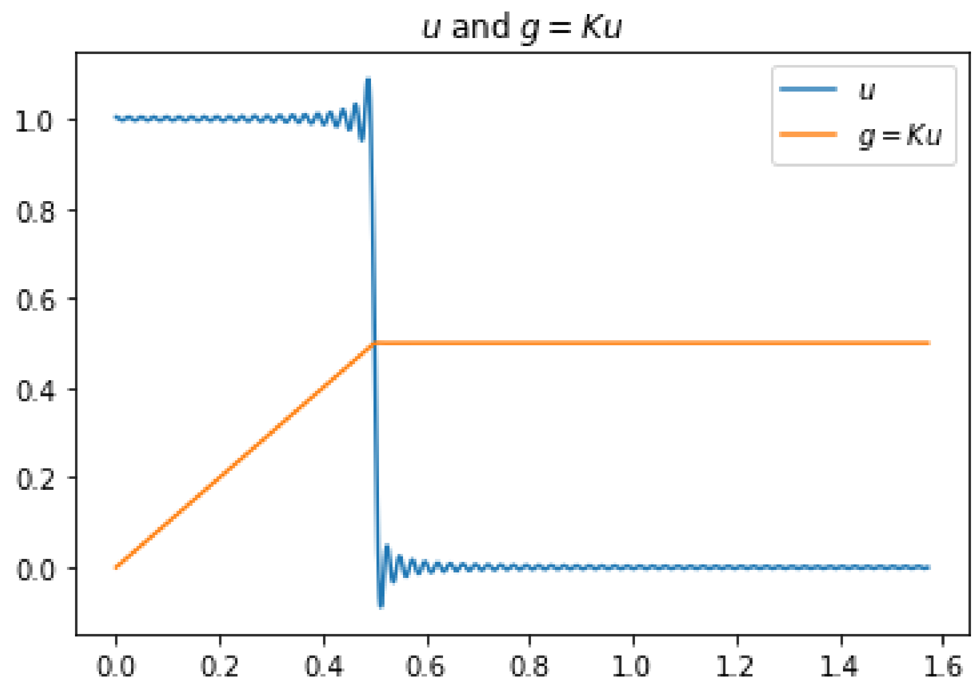
```
for k in range(0,M):
    sigma[k]=1/(2*k+1)
    koeff[k]=np.sinc(a*(2*k+1)/math.pi)*a*2/math.sqrt(math.pi)
    gkoeff[k]=koeff[k]*sigma[k]

def skalprod(k1,k2):
    N=1024
    h=math.pi/2/N
    summe=0
    for i in range(0,N):
        summe=summe+vk(k1,i*h)*vk(k2,i*h)
    summe=h*summe
    return summe

print(skalprod(3,5))
print(skalprod(4,4))
```

```
-0.0009765624999999355
0.9990234375000024
```

OK, at least they are orthonormal. Now let us check u and Ku.

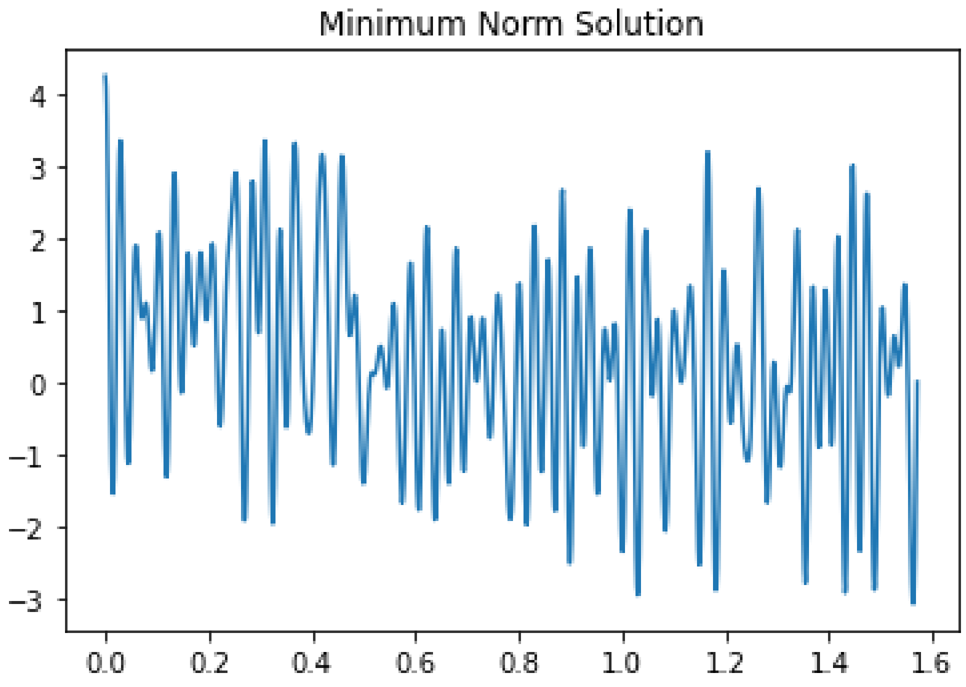
```
[8]: def evaluate(koeff,f,x):
         summe=0
         for k in range(0,koeff.size):
             summe=summe+koeff[k]*f(k,x)
         return summe

     def evalvec(koeff,f,X):
         N=X.size
         Y=np.zeros(N)
         for i in range(0,X.size):
             x=X[i]
             Y[i]=evaluate(koeff,f,x)
         return Y

     N=1024
     X=np.linspace(0,math.pi/2,N)
     U=evalvec(koeff,uk,X)
     KU=evalvec(gkoeff,vk,X)
     plt.plot(X,U,X,KU)
     plt.title('$u$ and $g=Ku$')
     plt.legend(['$u$','$g=Ku$'])
```

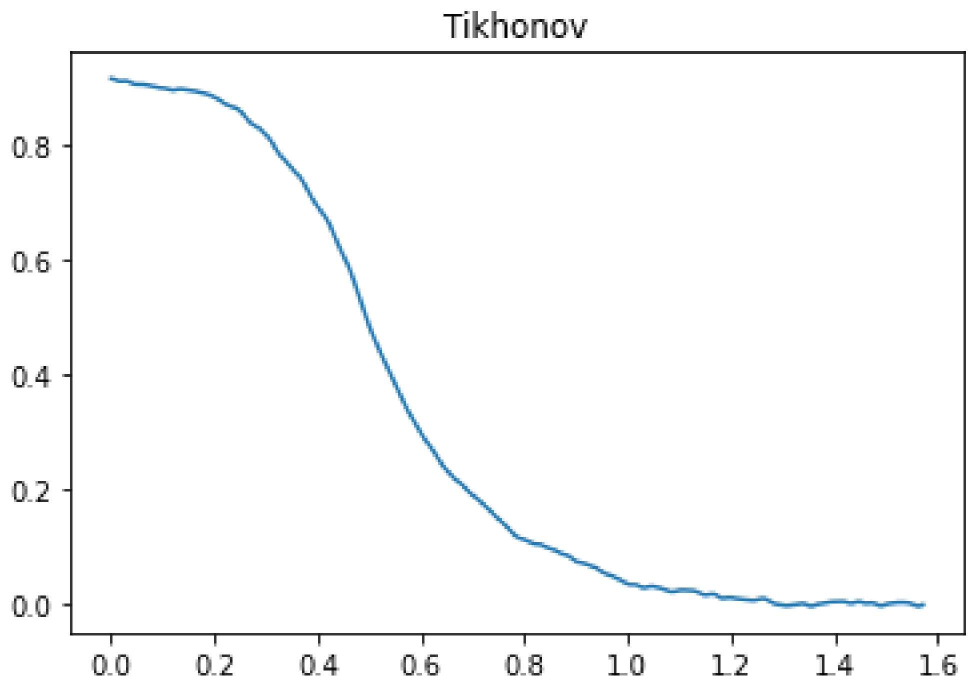
```
[8]: <matplotlib.legend.Legend at 0x7f5b0452b610>
```



[9]:
```
def Kalphaplus(koeff,g,alpha):
    N=koeff.size
    u=np.zeros(N)
    for i in range(0,N):
        u[i]=koeff[i]*g(alpha,sigma[i])
    return u
def gplus(alpha,sigma):
    return 1/sigma
def gplustikh(alpha,sigma):
    return sigma/(alpha*alpha+sigma*sigma)
def Kplus(koeff):
    return Kalphaplus(koeff,gplus,0)
def Kplustikh(koeff):
    return Kalphaplus(koeff,gplustikh,2e-1)

dev=1e-3
n = np.random.normal(0, dev, gkoeff.size)
gnoisy=gkoeff+n

U1=evalvec(Kplus(gnoisy),uk,X)
U2=evalvec(Kplustikh(gnoisy),uk,X)

plt.plot(X,U1)
plt.title('Minimum Norm Solution')
plt.figure()
plt.plot(X,U2)
plt.title('Tikhonov')
```

```
[9]: Text(0.5, 1.0, 'Tikhonov')
```




Tikhonov
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6

```
[ ]:
```

Discrete Fourier Transform

January 31, 2021

1 Discrete and analytic Fourier Transform

We start with a simple one-dimensional example. Assume that $f : \mathbb{R} \mapsto \mathbb{R}$ is a function with compact support on $[0, 2\pi]$. We want to compute numerical approximations of the analytic Fourier Transform of f for some arguments using the trapezoidal rule.

Discretizing the analytic Fourier transform on $n+1$ equidistant points with the trapezoidal rule (and observing that $f(0) = f(2\pi) = 0$) we have for $x_j = j\frac{2\pi}{n}$, $f_j = f(x_j)$, $j = 0 \dots n-1$:

$$\widehat{f}(\xi) = \frac{1}{\sqrt{2\pi}} \int_0^{2\pi} f(x) e^{-ix\xi}\, dx \sim \frac{\sqrt{2\pi}}{n} \sum_{j=0}^{n-1} f_j e^{-i\xi x_j}$$

The usual definition of the discrete Fourier Transform from numerical analysis is

$$\widehat{f}_k = \sum_{j=0}^{n-1} f_j e^{-2\pi ikj/n} = \sum_{j=0}^{n-1} f_j e^{-ikx_j}, k = 0 \dots n-1.$$

Note that $\widehat{f}_k = \widehat{f}_{k+n}$. Choosing $\xi = k$ we have

$$\widehat{f}(k) \sim \frac{\sqrt{2\pi}}{n} \widehat{f}_k$$

so we can compute approximations to the analytic Fourier transform using the discrete Fourier Transform (at $\xi = k$).

Now set $f_P(x) = f(x)$ on $[0, 2\pi]$ and $f_P(x + 2\pi) = f_P(x)$ for all x (periodic continuation). Then we have for the Fourier Coefficients of this function

$$c_k = \frac{1}{2\pi} \int_0^{2\pi} f(x) e^{-ikx}\, dx \sim \frac{1}{\sqrt{2\pi}} \widehat{f}(k) \sim \frac{1}{n} \widehat{f}_k.$$

Both approximations are not only valid for $k = 0 \dots n-1$ (that's where FFT computes the sum), but this is true for **all** $k \in Z$. However, this cannot be true. The left hand side in both cases is the evaluation of a Fourier transform, which we expect to go to zero fast with k (faster than any polynomial if $f \in \mathcal{S}$), but the right hand side is obviously periodic: $\widehat{f}_k = \widehat{f}_{k+n}$ and does not even go to zero.

We will see the true reason in Shannon's sampling theorem. However, we can easily demonstrate the problem. Now define $f_l(x) = e^{-ilx}$. Then we have

$$f_{kn}(x_j) = e^{in2\pi knj/n} = e^{2\pi ikj} = 1 = f_0(x).$$

This means that for all our sample points, all the $f_{kn}(x_j)$ are exactly the same. So when computing f_0: c_0, c_n, c_{-n} and so on all contribute to the result. In exactly the same way, when computing f_1: c_{n+1}, c_{-n+1} and so on contribute. Inserting the complete Fourier Series, we find that in fact the true approximation is

$$\sum_j c_{jn+k} \sim \frac{1}{n}\widehat{f}_k$$

Now it seems that we're lost - from our discretization, we can only compute the sum, not an individual c_k.

However, at this point we make use of the fact that $\widehat{f}(k)$ and thus c_k is rapidly decreasing. This means that if n is large enough, the coefficient with the order that is closed to 0 is the largest (by far if f is smooth). Example: $\widehat{f}_0$ approximates $c_0 + c_n + c_{-n} + \ldots$, but $c_n \sim 0$ etc, so $\widehat{f}_0$ is an approximation for c_0, as we suspected right from the beginning. $\widehat{f}_n = \widehat{f}_0$ is not an approximation for c_n (but again for c_0). Another example: $\widehat{f}_{n-1}$ approximates $c_{n-1} + c_{-1} + \ldots$. The largest one by far is c_{-1}, so $\widehat{f}_{n-1}$ approximates c_{-1}. Note that I left out the constants here for simplicity.

All in all, we find that

$$\frac{1}{\sqrt{2\pi}}\widehat{f}(k) \sim c_k \sim \sum_j c_{jn+k} \sim \frac{1}{n}\widehat{f}_k,\ k = -m \ldots m,\ n = 2m+1$$

where the approximation is good for $|k|$ small, and dubious for $|k| \sim m$.

With all this in mind, let us now look at the classical definition of discrete Fourier Transform. We will not use the fast transform at this point, since python and most implementations use a different definition of the FT.

Our example is the characteristic function of the interval $[-1+\pi, 1+\pi]$. Since this is a shifted version of our sinc example, using the computation rules we find that

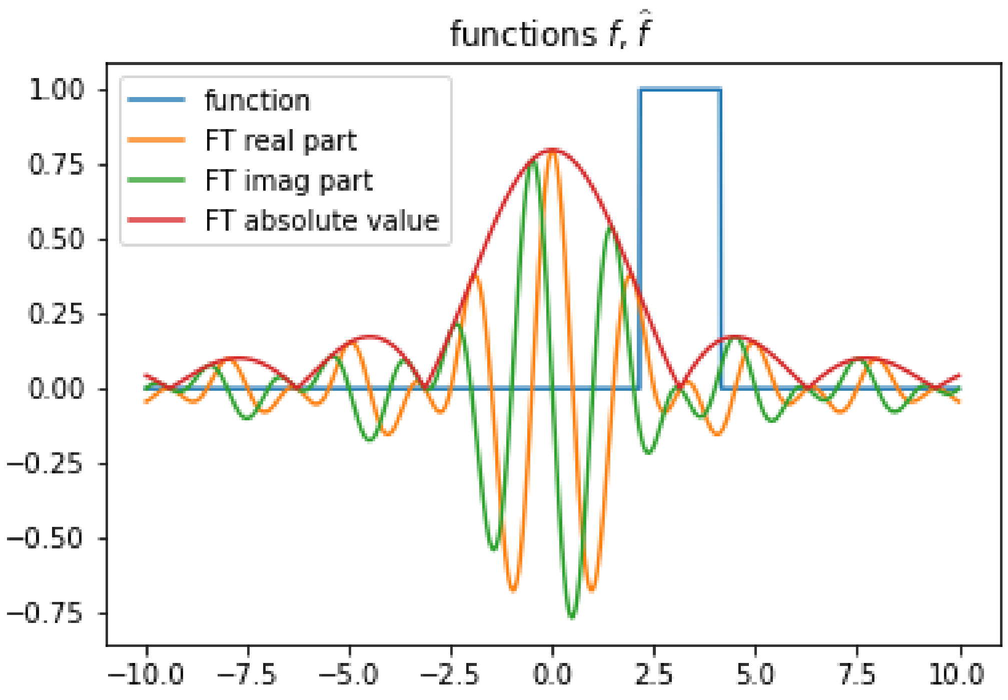
$$\widehat{f}(\xi) = e^{-i\pi\xi}\sqrt{\frac{2}{\pi}}sinc(\xi).$$

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math

# numpy has a different definition of the sinc function
def sinc(x):
    return np.sinc(x/math.pi)
```

[2]:
```
# use definition to slowly compute f_j hat.
def ft(Y,j):
    sum=0
    n=len(Y)
    for k,y in enumerate(Y):
        sum=sum+y*np.exp(-1j*k*j*2*math.pi/n)
    return sum
```

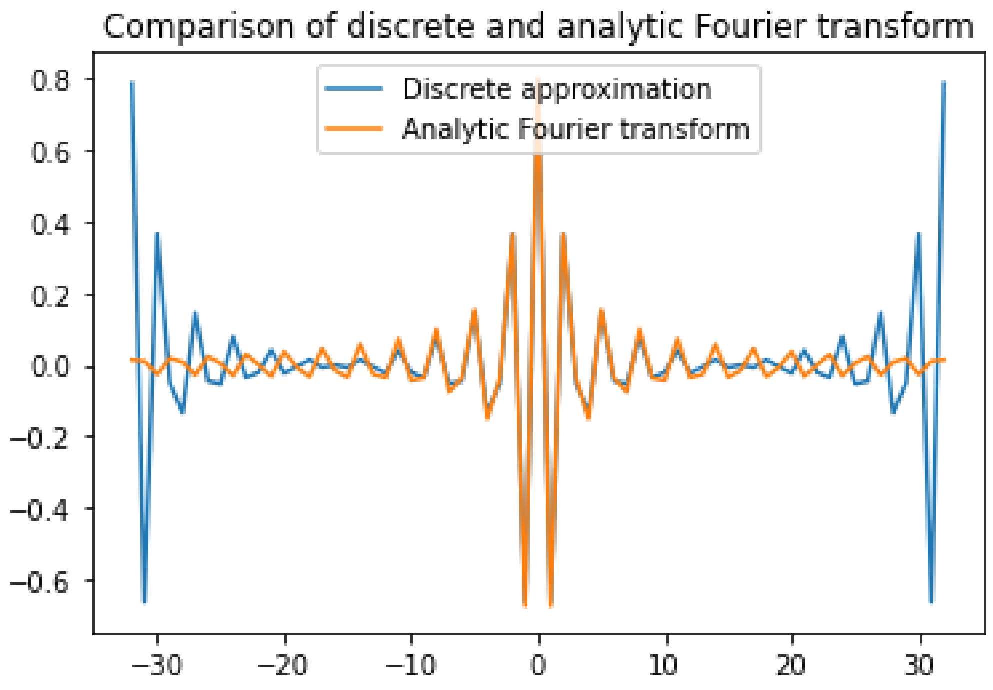
[3]:
```
def f(x):
    y=np.zeros(x.shape)
    I=np.where((x>-1+math.pi) & (x<1+math.pi))
    try:
        y[I]=1
    except IndexError:
        if (len(I)==0):
            return 0
        else:
            return 1
    return y

def fdach(xi):
    return sinc(xi)*math.sqrt(2/math.pi) *np.exp(-xi*1j*math.pi)
```

[4]:
```
N=1024
Xplot=np.linspace(-10,10,N)
Yplot=fdach(Xplot)
plt.plot(Xplot,f(Xplot),Xplot,np.real(Yplot),Xplot,np.imag(Yplot),Xplot,np.
  ↪abs(Yplot))
plt.title('functions $f$, $\\widehat f$')

plt.legend(['function', 'FT real part', 'FT imag part', 'FT absolute value']);
```



```
[5]: n=32
     X=np.linspace(0,2*math.pi,n)
     Y=f(X)
     xi=np.arange(-n,n+1)
     Ydach=ft(Y,xi)*math.sqrt(2*math.pi)/n
     Fdach=fdach(xi)
     plt.plot(xi,np.real(Ydach),xi,np.real(Fdach))
     plt.legend(['Discrete approximation','Analytic Fourier transform'])
     plt.title('Comparison of discrete and analytic Fourier transform')
```

```
[5]: Text(0.5, 1.0, 'Comparison of discrete and analytic Fourier transform')
```



This is exactly what we expected. The discrete approximation is periodic with period length n. The approximation $\widehat{f}(k) \sim \frac{\sqrt{2\pi}}{n}\widehat{f}_k$ is good for $|k|$ small, acceptable up to $|k|/2$, and completely off for $|k| > n/2$.

In the ordinary Fourier transform, we number the Fourier coefficients $\widehat{f}_0, \ldots, \widehat{f}_{n-1}$, and find that e.g. $\widehat{f}_1 \sim c_1$, but $\widehat{f}_{n-1} \sim c_{-1}$. This is not very convenient. Rather, we would like to number $\widehat{f}_{-m} \ldots \widehat{f}_m$, $n = 2m+1$ (we take n odd for simplicity here).

However: We have a very nice symmetry between the Fourier Transform and its inverse. This gets lost if we number differently, so we also number $f_{-m}, \ldots f_m$. Since we still want to use our considerations from above, we still need $f_k = f(2\pi k/n)$, so f should now have its support centered at zero in the interval $[-\pi, \pi]$ (which is nicer anyway). Putting all this together, we arrive at a much more natural definition of the discrete Fourier Transform (sometimes called centered transform):

$$\widehat{f}_k = \sum_{j=-m}^{m} f_j e^{-2\pi i k j/n}, \; k = -m \ldots m$$

for $n = 2m+1$ odd and

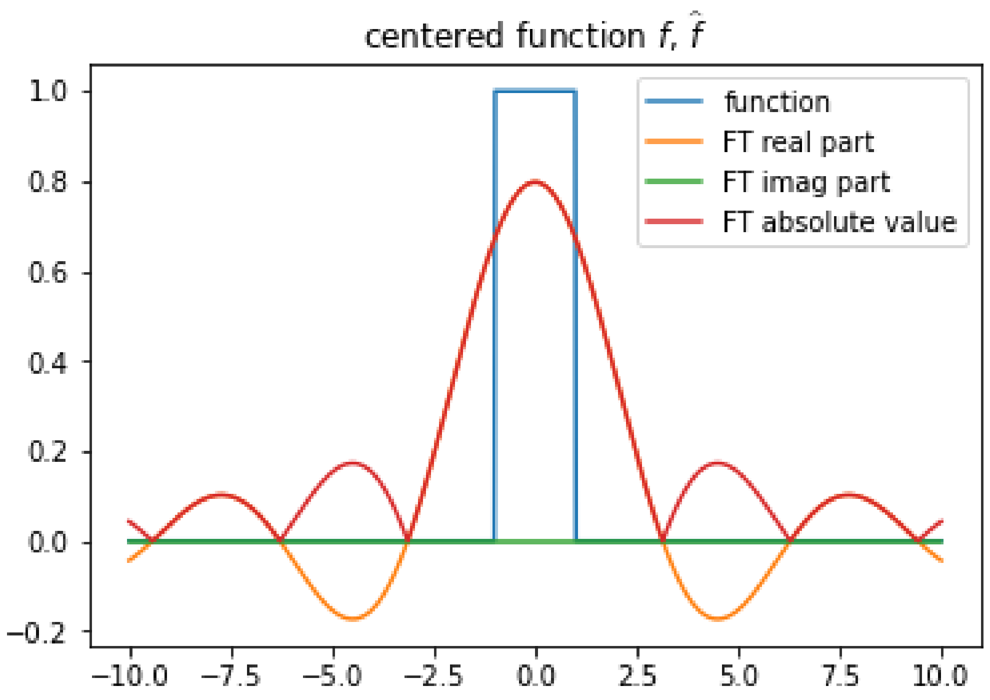
$$\widehat{f}_k = \sum_{j=-m}^{m-1} f_j e^{-2\pi i k j/n}, \; k = -m \ldots m-1$$

for $n = 2m$ even. From our considerations we now simply have $\frac{\sqrt{2\pi}}{n}\widehat{f}_k \sim \widehat{f}(k)$.

We implement this and check as above, this time using the example of the characteristic function of $[-1,1]$.

[6]:
```
def f_centered(x):
    y=np.zeros(x.shape)
    I=np.where((x>-1) & (x<1))
    try:
        y[I]=1
    except IndexError:
        if (len(I)==0):
            return 0
        else:
            return 1
    return y

def f_centereddach(xi):
    return sinc(xi)*math.sqrt(2/math.pi)
```

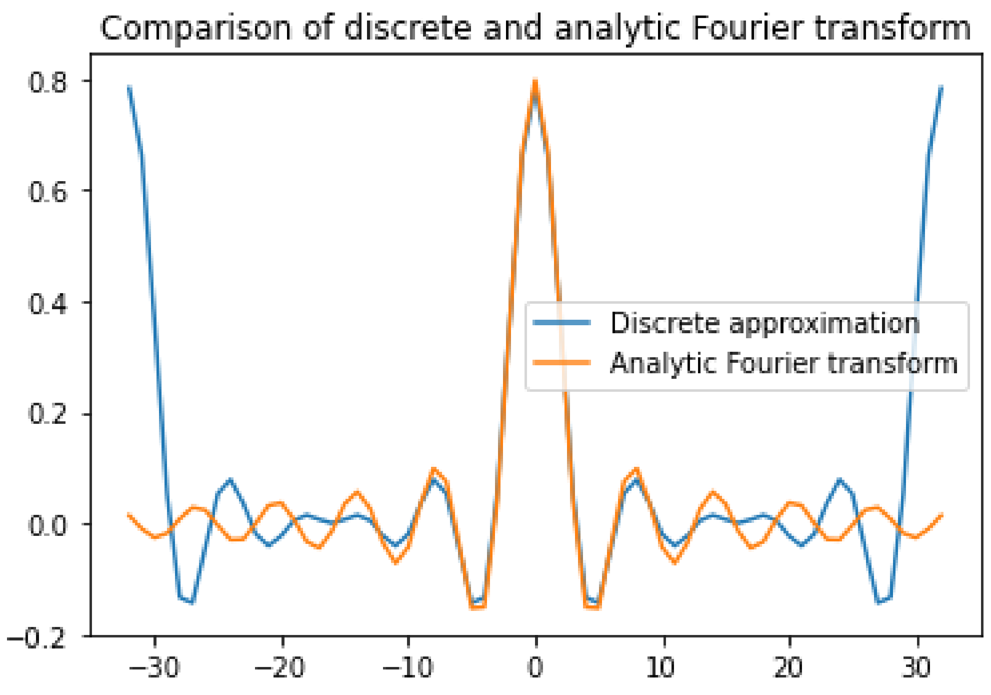
[7]:
```
Yplot=f_centereddach(Xplot)
plt.plot(Xplot,f_centered(Xplot),Xplot,np.real(Yplot),Xplot,np.
 ↪imag(Yplot),Xplot,np.abs(Yplot))
plt.title('centered function $f$, $\\widehat f$')
plt.legend(['function', 'FT real part', 'FT imag part', 'FT absolute value']);
```



```
[8]: # use definition to slowly compute centered f_j hat.
     def ft_centered(Y,j):
         sum=0
         n=len(Y)
         m=len(Y)//2
         for k,y in enumerate(Y):
             sum=sum+y*np.exp(-1j*(k-m)*j*2*math.pi/n)
         return sum
```

```
[9]: n=32
     X=np.linspace(-math.pi,math.pi,n)
     Y=f_centered(X)
     xi=np.arange(-n,n+1)
     Ydach=ft_centered(Y,xi)*math.sqrt(2*math.pi)/n
     Fdach=f_centereddach(xi)
     plt.plot(xi,np.real(Ydach),xi,np.real(Fdach))
     plt.legend(['Discrete approximation','Analytic Fourier transform'])
     plt.title('Comparison of discrete and analytic Fourier transform')
     #print(müreal(ft(Y,2))*math.sqrt(2*math.pi)/n,fdach(2))
```

```
[9]: Text(0.5, 1.0, 'Comparison of discrete and analytic Fourier transform')
```



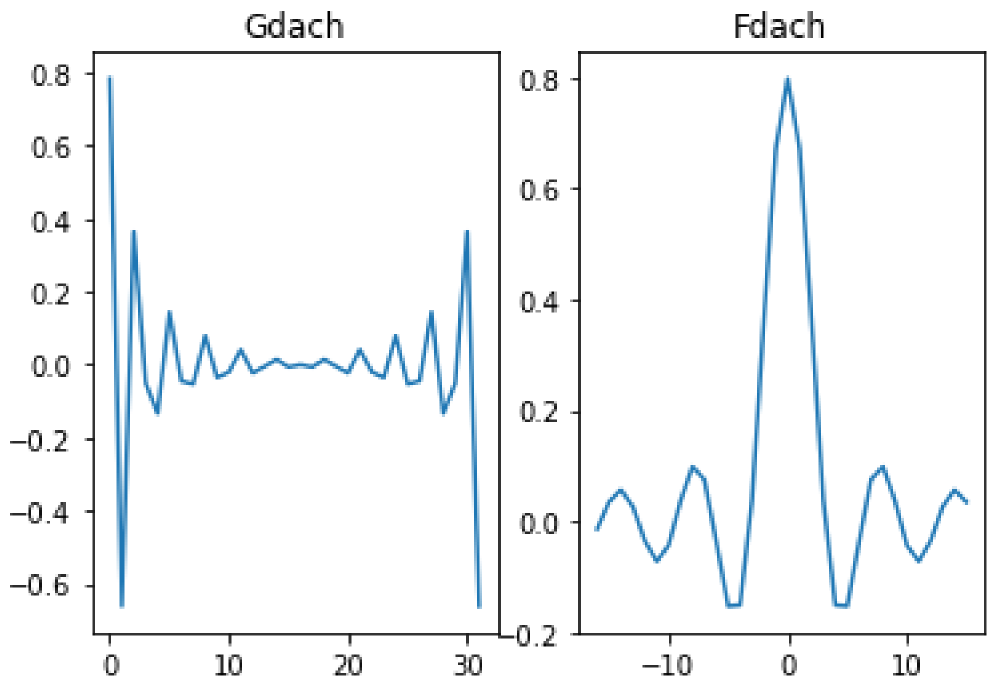
There's a simple relation between the two transforms. Assume that f_k is numbered $k = -m \dots m$, $g_k = f_{k-m}$ numbered $k = 0 \dots n-1$. Then for the centered transform (and for simplicity n odd):

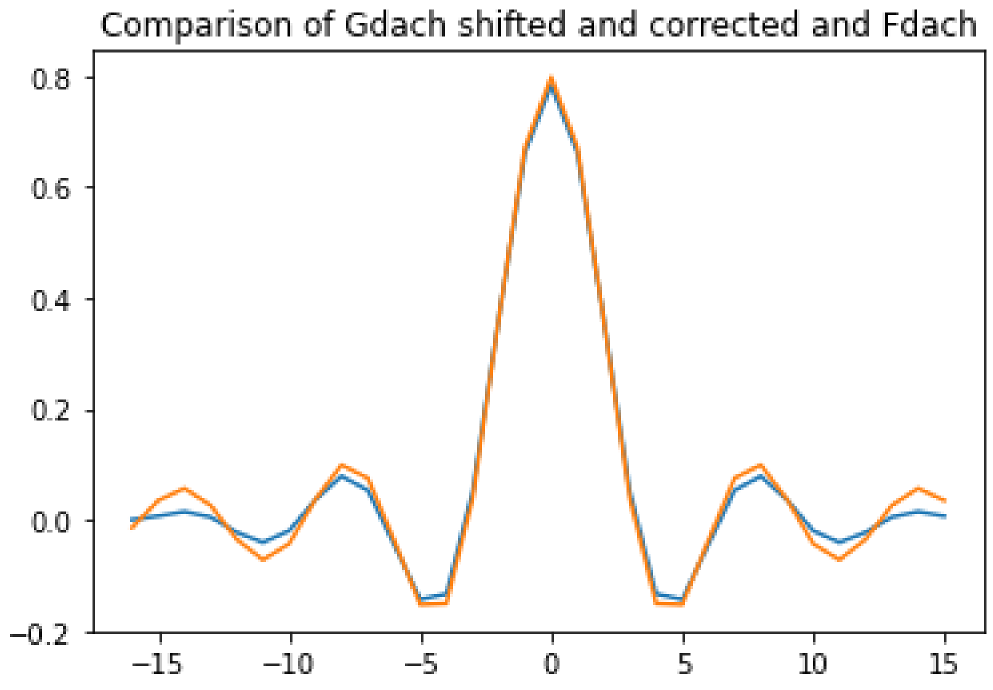
$$\begin{aligned}
\widehat{f_k} &= \sum_{j=-m}^{m} f_j e^{-2\pi ikj/n} \\
&= \sum_{j=-m}^{m} g_{j+m} e^{-2\pi ikj/n} \\
&= \sum_{j=0}^{n-1} g_j e^{-2\pi ik(j-m)/n} \\
&= e^{-2\pi ikm/n} \sum_{j=0}^{n-1} g_j e^{-2\pi ikj/n} \\
&= e^{2\pi ikm/n} \widehat{g}_k
\end{aligned}$$

where the Fourier Transform on g is the ordinary one. Exactly the same is true for n even (and in this case the prefactor is just $(-1)^k$). Viewing $(\widehat{f_k})$ and $(\widehat{g}_k)$ as vectors, we have $(\widehat{f}_{-m}, \ldots f_0, \ldots f_m)$ and $(g_0, \ldots, g_{n-1})$. Since we need to compute f_0 from g_0 and so on, to compute f from g, the second vector must be rotated/shifted by m indices before multiplication. This is exactly what the function fftshift does in Python and Matlab.

[10]:
```
n=32
m=n//2
X=np.linspace(-math.pi,math.pi,n)
Y=f_centered(X)
#Y.fill(0)
#Y[m+1]=1
xi=np.arange(-m,m+n%2)
Fdach=ft_centered(Y,xi)*math.sqrt(2*math.pi)/n
Fdach=f_centereddach(xi)

xi=np.arange(0,n)
Gdach=ft(Y,xi)*math.sqrt(2*math.pi)/n
plt.subplot(121)
plt.plot(xi,Gdach.real)
plt.title('Gdach')
plt.subplot(122)
plt.plot(xi-m,Fdach.real)
plt.title('Fdach')
plt.figure()
corr=np.exp(2*math.pi*1j *xi *m/n)
plt.plot(xi-m,np.fft.fftshift(corr*Gdach).real,xi-m,Fdach.real)
plt.title('Comparison of Gdach shifted and corrected and Fdach')
plt.figure()
```

[10]:
```
<Figure size 432x288 with 0 Axes>
```

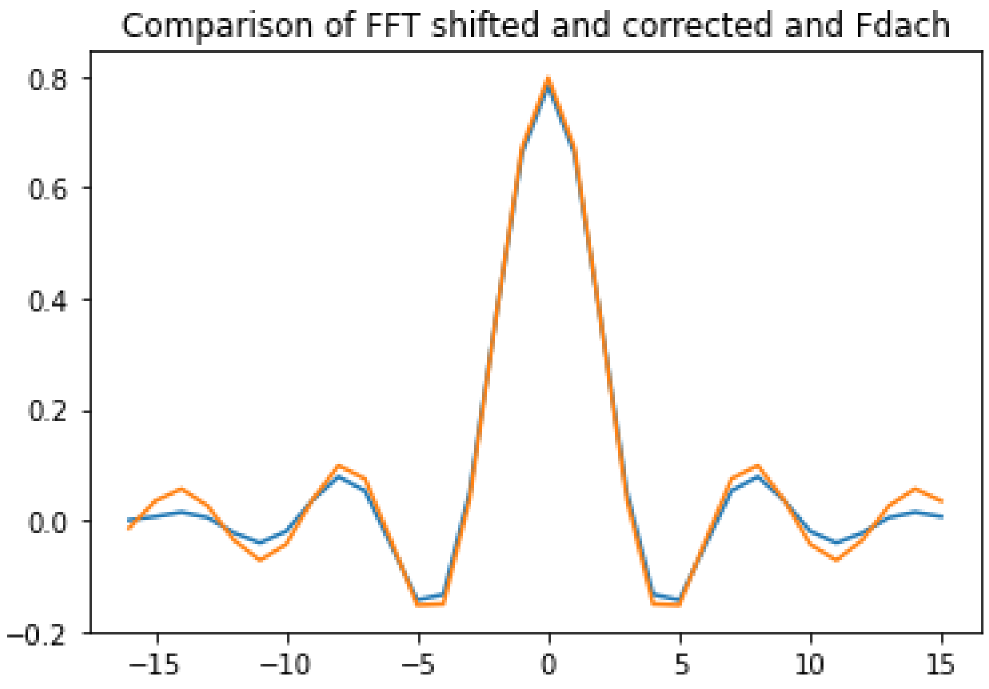




```
<Figure size 432x288 with 0 Axes>
```

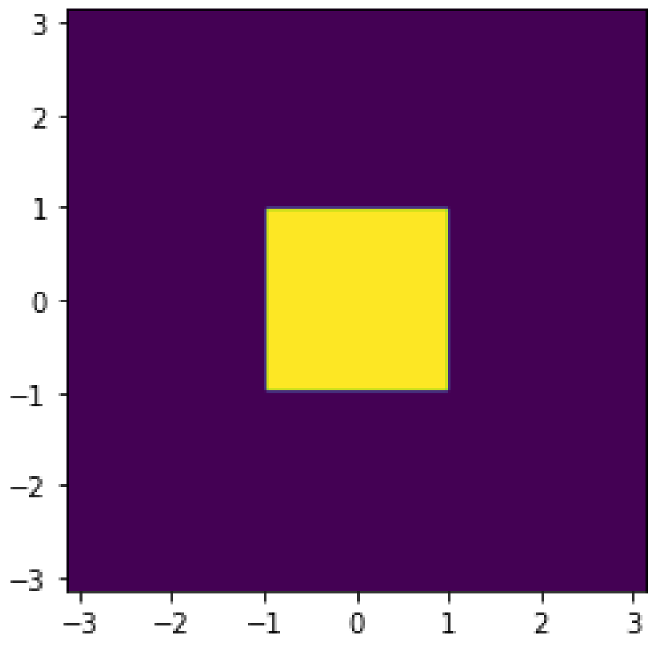
Finally, we now use FFT for the computation. This is the routine we will really use for computing approximations to the analytic Fourier Transform.

[11]:
```
def centered_fft(Y):
    n=len(Y)
    m=n//2
    Z=np.fft.fft(Y)*math.sqrt(2*math.pi)/n
    corr=np.exp(2*math.pi*1j *xi *m/n)
    return np.fft.fftshift(corr*Z)

Z=centered_fft(Y)
plt.plot(xi-m,Z.real,xi-m,Fdach.real)
plt.title('Comparison of FFT shifted and corrected and Fdach');
```

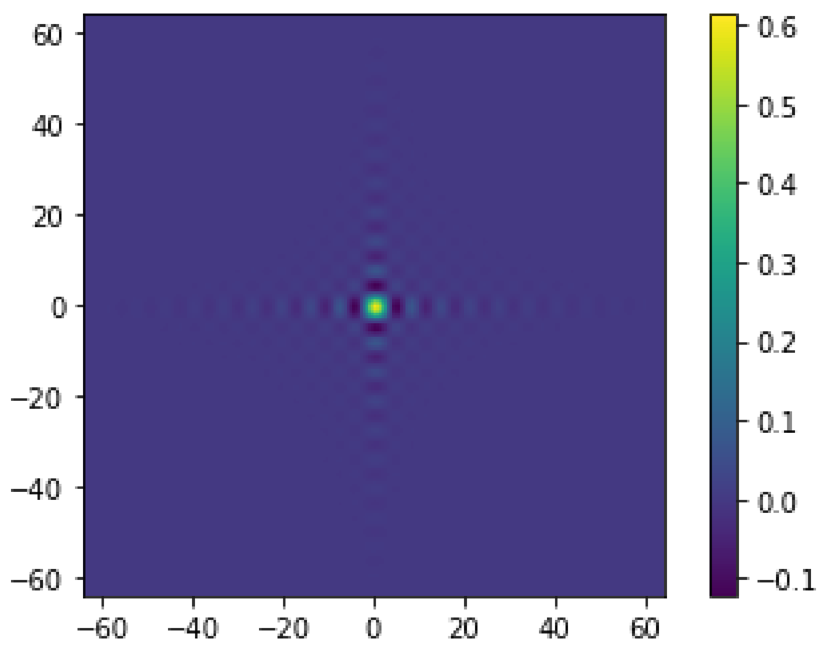


[12]:
```
N=128
X=np.linspace(-math.pi,math.pi,N)
Y=np.linspace(-math.pi,math.pi,N)
(Xgrid,Ygrid)=np.meshgrid(X,Y)
f=np.zeros(Xgrid.shape)
I=np.where((np.abs(Xgrid)<1)&(np.abs(Ygrid)<1))
f[I]=1
extent=[-math.pi,math.pi,-math.pi,math.pi]
plt.imshow(f,extent=extent)
```

[12]:
```
<matplotlib.image.AxesImage at 0x7fc43a4c4e50>
```



[13]:
```
def ft2_centered(f):
    fdach=np.fft.fft2(f)*2*math.pi/(N*N)
    fdach=np.fft.fftshift(fdach)
    for i in range(0,f.shape[0]):
        for k in range(0,f.shape[1]):
            fdach[i,k]*=(-1)**(i+k)
    return fdach

f2dach=ft2_centered(f)
extent2=(-N/2,N/2,-N/2,N/2)
plt.imshow(f2dach.real,extent=extent2)
plt.colorbar();
```


60
40
20
0
−20
−40
−60
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−0.1

```
[ ]:
```

Aufgabe 4

January 31, 2021

$$
\begin{aligned}
(\frac{\partial}{\partial x_i}\chi_K)(\varphi) &= -\int_{\mathbb{R}^n} \chi_K(x)\,(\frac{\partial}{\partial x_i}\varphi)(x)\,dx \\
&= -\int_K (\frac{\partial}{\partial x_i}\varphi)(x)\,dx \\
&= -\int_{\partial K} \nu_i(x)\,\varphi(x)\,dx \\
&= -\int_{\mathbb{R}^n} \nu_i(x)\delta_{\partial K}(x)\,\varphi(x)\,dx \\
&= -\nu_i(\cdot)\delta_{\partial K}(\cdot)
\end{aligned}
$$

```
[ ]:
```