전산천문학 HW3 Solution

April 29, 2020

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.odr import *
```

1. Numerica Integration of $\sin(x^4)$

[2]:
```
x1,x2=0.,4.
def sin4(x):
    return np.sin(np.power(x,4.))
```

(a) Composite Trapezoidal Rule

[3]:
```
int_prev,error,TOL=1.,1.,1.0e-8
int_history=[]
N=0

while (error>TOL):
    N+=100
    dx_i=(x2-x1)/N
    x_i=np.arange(x1,x2+dx_i,dx_i)
    int_i=dx_i*(np.sum(sin4(x_i))-0.5*sin4(x1)-0.5*sin4(x2))
    error=np.abs(int_i-int_prev) # 이전의 적분값과 비교하여 error값을 계산
    if (N%500==0):
        print ('N={:d},I={:.6f}, error={:.2e}'.format(N,int_i,error)) # 중간 과정
    int_prev=int_i
    int_history.append(int_i)

print ('N={:d},I={:.6f} error={:.2e}'.format(N,int_i,error)) # 최종 결과
```

```
N=500,I=0.346972, error=3.98e-05
N=1000,I=0.347018, error=3.35e-06
N=1500,I=0.347026, error=9.12e-07
N=2000,I=0.347029, error=3.71e-07
N=2500,I=0.347030, error=1.86e-07
N=3000,I=0.347031, error=1.06e-07
N=3500,I=0.347031, error=6.64e-08
```

```
N=4000,I=0.346055, error=9.77e-04
N=4500,I=0.347031, error=3.09e-08
N=5000,I=0.347032, error=2.24e-08
N=5500,I=0.346312, error=1.30e-05
N=6000,I=0.347032, error=1.29e-08
N=6500,I=0.347032, error=1.01e-08
N=7000,I=0.347032, error=5.76e-04
N=7100,I=0.347032 error=7.76e-09
```

(b) Composite Simpson's Rule

[4]:
```python
int_prev,error,TOL=1.,1.,1.0e-8
int_history=[]
N=0

while (error>TOL):
    N+=10
    dx=(x2-x1)/N
    x_p,x_q=np.arange(x1,x2+dx,2*dx),np.arange(x1+dx,x2,2*dx)
    int_i=(dx/3.)*(2.*np.sum(sin4(x_p))+4.*np.sum(sin4(x_q))-sin4(x1)-sin4(x2))
    error=np.abs(int_i-int_prev)
    if (N%100==0):
        print ('N={:d},I={:.6f} error={:.2e}'.format(N,int_i,error))
    int_prev=int_i
    int_history.append(int_i)

print ('N={:d},I={:.6f} error={:.2e}'.format(N,int_i,error))
```

```
N=100,I=0.609184 error=5.80e-01
N=200,I=0.301175 error=8.73e-02
N=300,I=0.400661 error=4.46e-02
N=400,I=0.347462 error=1.84e-04
N=500,I=0.347084 error=8.20e-06
N=600,I=0.347049 error=1.67e-06
N=700,I=0.347039 error=5.57e-07
N=800,I=0.347036 error=2.37e-07
N=900,I=0.347034 error=1.17e-07
N=1000,I=0.347033 error=6.38e-08
N=1100,I=0.347033 error=3.74e-08
N=1200,I=0.347033 error=2.32e-08
N=1300,I=0.347033 error=1.50e-08
N=1400,I=0.347032 error=1.01e-08
N=1410,I=0.347032 error=9.75e-09
```

(c) Gaussian Quadrature

[5]:
```
def sin4_gq(t):
    return 2.*np.sin(np.power((2.+2.*t),4.)) # 적분 범위가 -1에서 1이 되도록 식을 변경
```

[6]:
```
int_prev,error,TOL=1.,1.,1.0e-7
int_history=[]
N=1

while (error>TOL):
    N*=2
    x,w=np.polynomial.legendre.leggauss(N)
    int_i=0
    for i in range(0,N):
        int_i+= w[i]*sin4_gq(x[i])
    error=np.abs(int_i-int_prev)
    int_prev=int_i
    print ('N={:d} nodes, I={:.6f} error={:.2e}'.format(N,int_i,error))
    int_history.append(int_i)
```

```
N=2 nodes, I=-1.015485 error=2.02e+00
N=4 nodes, I=1.177074 error=2.19e+00
N=8 nodes, I=0.155457 error=1.02e+00
N=16 nodes, I=-0.652353 error=8.08e-01
N=32 nodes, I=0.923506 error=1.58e+00
N=64 nodes, I=0.107042 error=8.16e-01
N=128 nodes, I=0.345722 error=2.39e-01
N=256 nodes, I=0.347032 error=1.31e-03
N=512 nodes, I=0.347032 error=1.06e-14
```

2. Limb Darkening

문제에서 주어진 식에서 적분 기호 내의 식을 $G(\phi,\mu)$라고 하자.

식을 보면 $G(\phi,\mu)$는 $\phi=0$에서 정의되지 않지만, 극한을 이용하여 어떤 값으로 수렴하는지 확인할 수 있다.

$$\lim_{\phi\to 0} G(\phi,\mu) = \lim_{\phi\to 0} \frac{\phi\tan^{-1}(\mu\tan\phi)}{1-\phi\cot\phi} d\phi \tag{1}$$

$$= \lim_{\phi\to 0} \frac{\phi^2\sin\phi}{\sin\phi-\phi\cos\phi}\cdot\frac{\tan^{-1}(\mu\tan\phi)}{\phi} \tag{2}$$

이 때 로피탈의 정리를 활용하면,

$$\lim_{\phi\to 0} \frac{\phi^2\sin\phi}{\sin\phi-\phi\cos\phi} = \lim_{\phi\to 0} \frac{2\phi\sin\phi+\phi^2\cos\phi}{\phi\sin\phi} = 3 \tag{3}$$

$$\lim_{\phi \to 0} \frac{\tan^{-1}(\mu \tan \phi)}{\phi} = \lim_{\phi \to 0} \frac{\mu \sec^2 \phi}{1 + (\mu \tan \phi)^2} = \mu \tag{4}$$

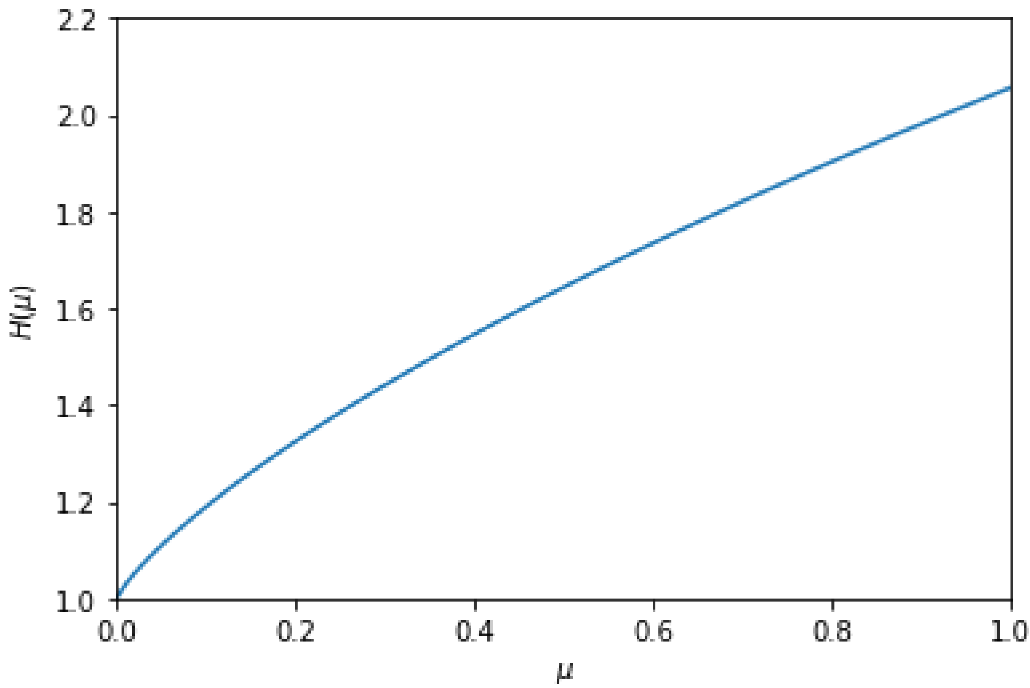
로 각각 두 식이 모두 $\phi \to 0$에서 수렴하기 때문에, $lim_{\phi \to 0} G(\phi, \mu) = 3\mu$가 된다.

[7]:
```
def G(phi,mu):
    if phi == 0:
        return 3*mu
    else:
        return phi*np.arctan(mu*np.tan(phi))/(1.-phi/np.tan(phi))
G=np.vectorize(G)
```

[8]:
```
def limb_dark(mu):
    phi1,phi2=0.,np.pi/2.
    N=10000
    dphi=(phi2-phi1)/N
    phi_p,phi_q=np.arange(phi1,phi2+dphi,2*dphi),np.arange(phi1+dphi,phi2,2*dphi)
    int_i=(dphi/3.)*(2.*np.sum(G(phi_p,mu))+4.*np.
 ↪sum(G(phi_q,mu))-G(phi1,mu)-G(phi2,mu))
    return np.exp(int_i/np.pi)/(1.+mu)
mu_range=np.linspace(0.,1.0,100)
H=[]
for mu_i in mu_range:
    H.append(limb_dark(mu_i))
```

[9]:
```
plt.plot(mu_range,H)
plt.xlim(0.,1.)
plt.ylim(1.0,2.2)
plt.xlabel(r'$\mu$')
plt.ylabel(r'$H(\mu)$')
```

[9]:
```
Text(0, 0.5, '$H(\\mu)$')
```



3. Interpolation

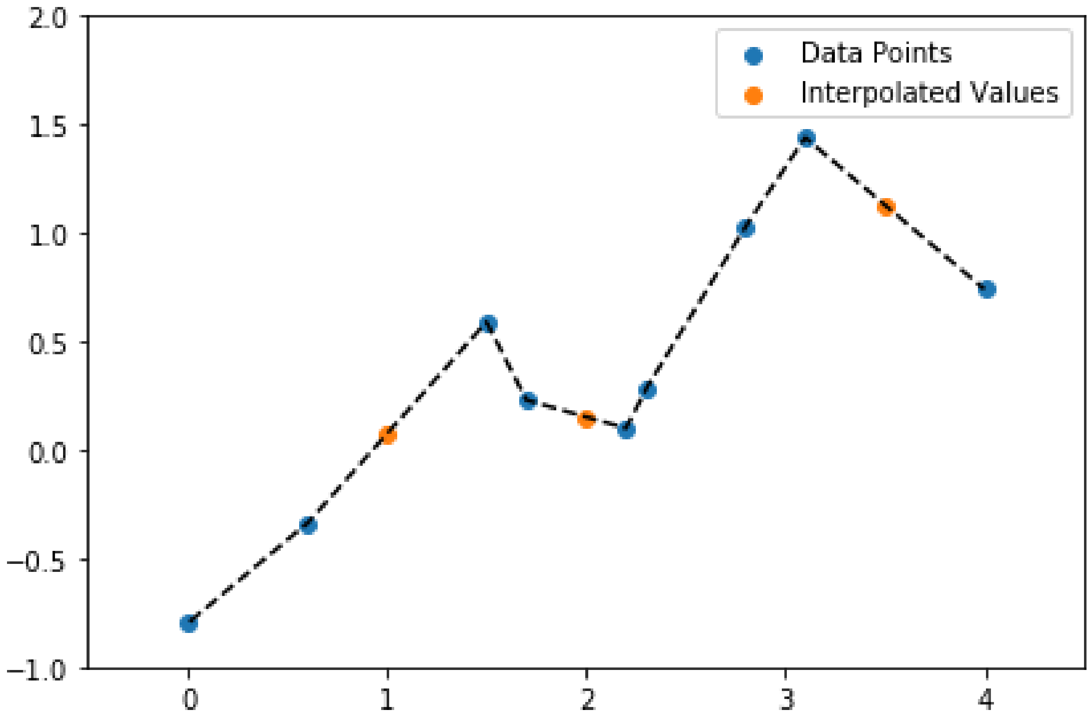
[10]:
```
data_x=np.array([0.,0.6,1.5,1.7,2.2,2.3,2.8,3.1,4.])
data_y=np.array([-0.8,-0.34,0.59,0.23,0.1,0.28,1.03,1.44,0.74])
interp_x=[1.,2.,3.5]
```

(a) Piecewise Linear Interpolation

[11]:
```
def lin_interp(x):
    x_in,x_out=data_x[x>data_x][-1],data_x[x<data_x][0] # x1,x2의 위치
    y_in,y_out=data_y[x>data_x][-1],data_y[x<data_x][0] # x1,x2에서의 y값
    return y_in+(y_out-y_in)*(x-x_in)/(x_out-x_in) # Linear Interpolation 진행
lin_interp=np.vectorize(lin_interp)

interp_y=lin_interp(interp_x)
x_range=np.arange(data_x[0]+0.01,data_x[-1],0.001)

for x_i in interp_x:
    print ('for x={:.3f}, the expected value is y={:.6f}'.
 ↪format(x_i,lin_interp(x_i)))

plt.figure(figsize=(6,4))
plt.scatter(data_x,data_y,label='Data Points')
```

```
plt.scatter(interp_x,interp_y,label='Interpolated Values')
plt.plot(x_range,lin_interp(x_range),c='k',ls='--')
plt.xlim(-0.5,4.5)
plt.ylim(-1.,2.)
plt.legend()
plt.tight_layout()
```

```
for x=1.000, the expected value is y=0.073333
for x=2.000, the expected value is y=0.152000
for x=3.500, the expected value is y=1.128889
```

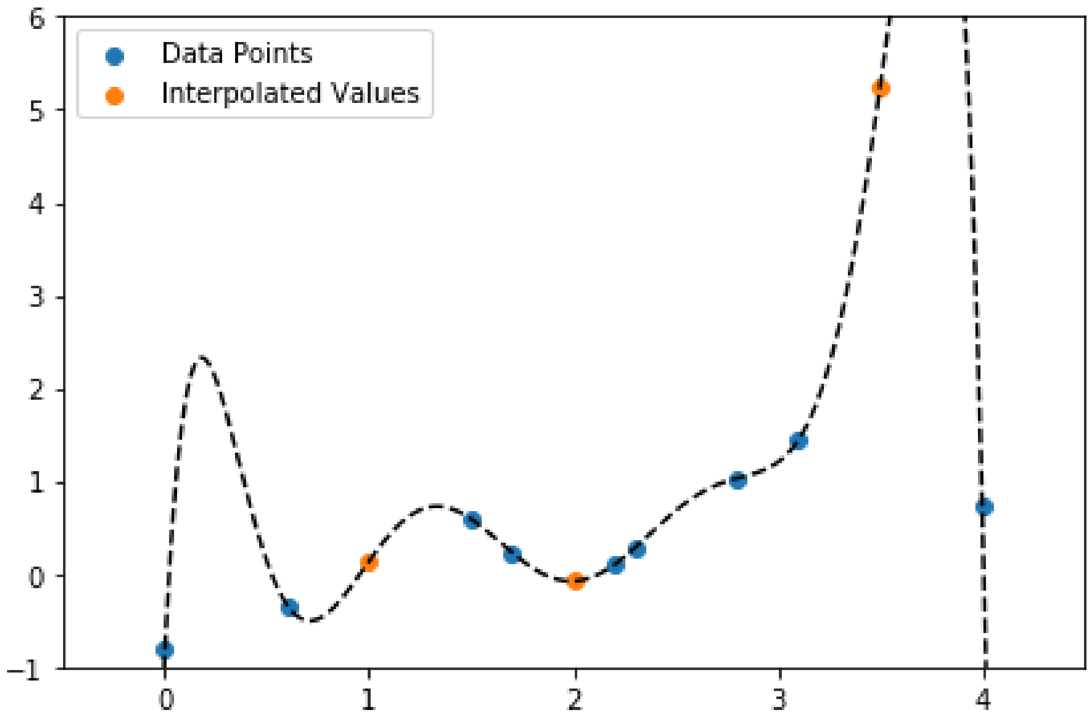


(b) 8th-Order Polynomial Interpolation

[12]:
```
p=np.polyfit(data_x,data_y,8)
poly=np.poly1d(p)
x_range=np.arange(-3.,5.,0.01)

for x_i in interp_x:
    print ('for x={:.3f}, the expected value is y={:.3f}'.format(x_i,poly(x_i)))

plt.figure(figsize=(6,4))
plt.scatter(data_x,data_y,label='Data Points')
plt.scatter(interp_x,poly(interp_x),label='Interpolated Values')
```

```
plt.plot(x_range,poly(x_range),c='k',ls='--')
plt.xlim(-0.5,4.5)
plt.ylim(-1.,6.)
plt.legend()
plt.tight_layout()
```

```
for x=1.000, the expected value is y=0.140
for x=2.000, the expected value is y=-0.078
for x=3.500, the expected value is y=5.227
```

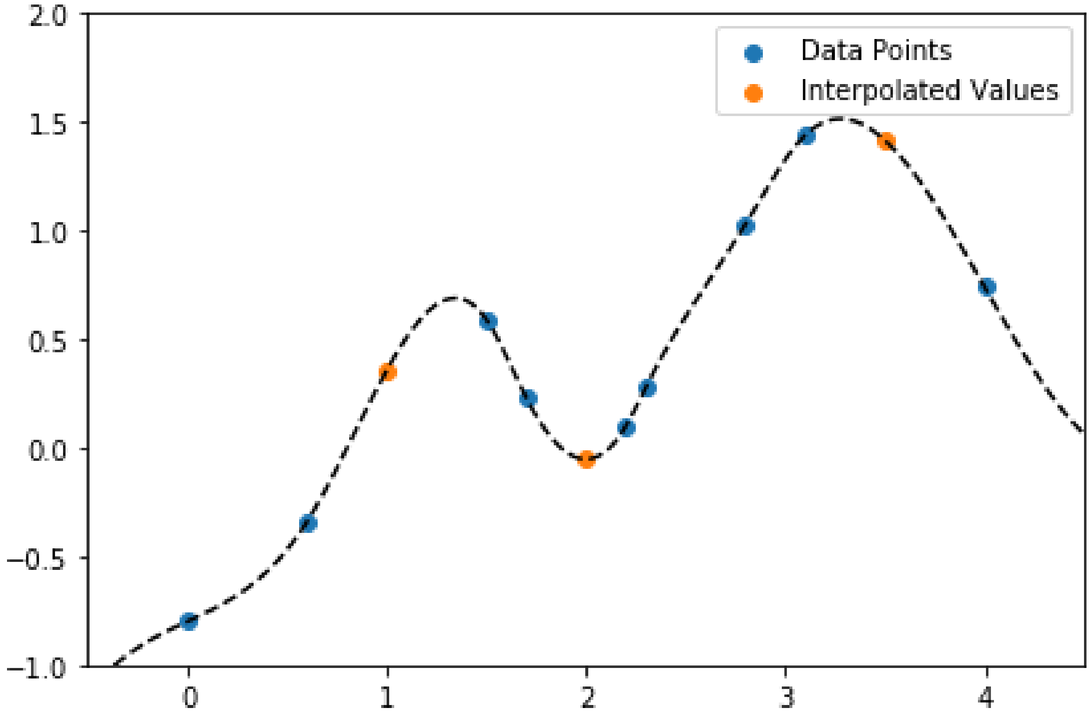


(c) Natural Cubic Spline Interpolation

[13]:
```
from scipy import interpolate
result=interpolate.CubicSpline(data_x,data_y,bc_type='natural')
x_range=np.arange(-3.,5.,0.01)

for x_i in interp_x:
    print ('for x={:.3f}, the expected value is y={:.3f}'.
 ↪format(x_i,result(x_i)))

plt.figure(figsize=(6,4))
plt.scatter(data_x,data_y,label='Data Points')
plt.scatter(interp_x,result(interp_x),label='Interpolated Values')
```

```
plt.plot(x_range,result(x_range),c='k',ls='--')
plt.xlim(-0.5,4.5)
plt.ylim(-1.,2.)
plt.legend()
plt.tight_layout()
```

```
for x=1.000, the expected value is y=0.358
for x=2.000, the expected value is y=-0.053
for x=3.500, the expected value is y=1.416
```

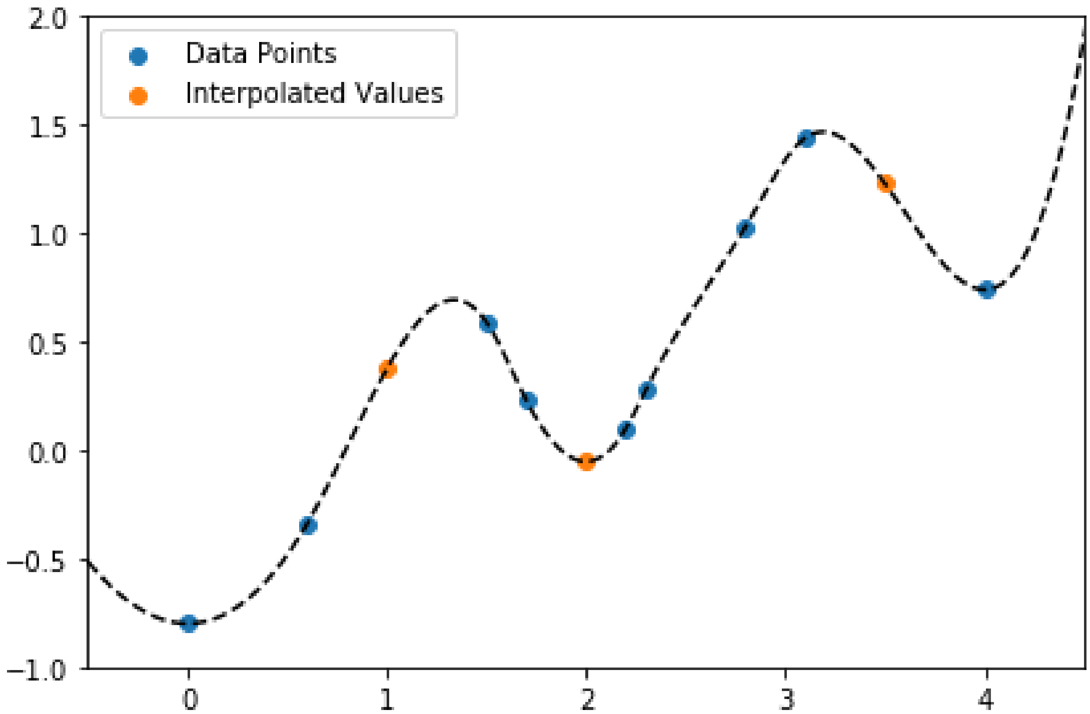


(d) Clamped Cubic Spline Interpolation

[14]:
```
from scipy import interpolate
result=interpolate.CubicSpline(data_x,data_y,bc_type='clamped')
x_range=np.arange(-3.,5.,0.01)

for x_i in interp_x:
    print ('for x={:.3f}, the expected value is y={:.3f}'.
 ↪format(x_i,result(x_i)))

plt.figure(figsize=(6,4))
plt.scatter(data_x,data_y,label='Data Points')
plt.scatter(interp_x,result(interp_x),label='Interpolated Values')
```

```
plt.plot(x_range,result(x_range),c='k',ls='--')
plt.xlim(-0.5,4.5)
plt.ylim(-1.,2.)
plt.legend()
plt.tight_layout()
```

```
for x=1.000, the expected value is y=0.375
for x=2.000, the expected value is y=-0.053
for x=3.500, the expected value is y=1.229
```

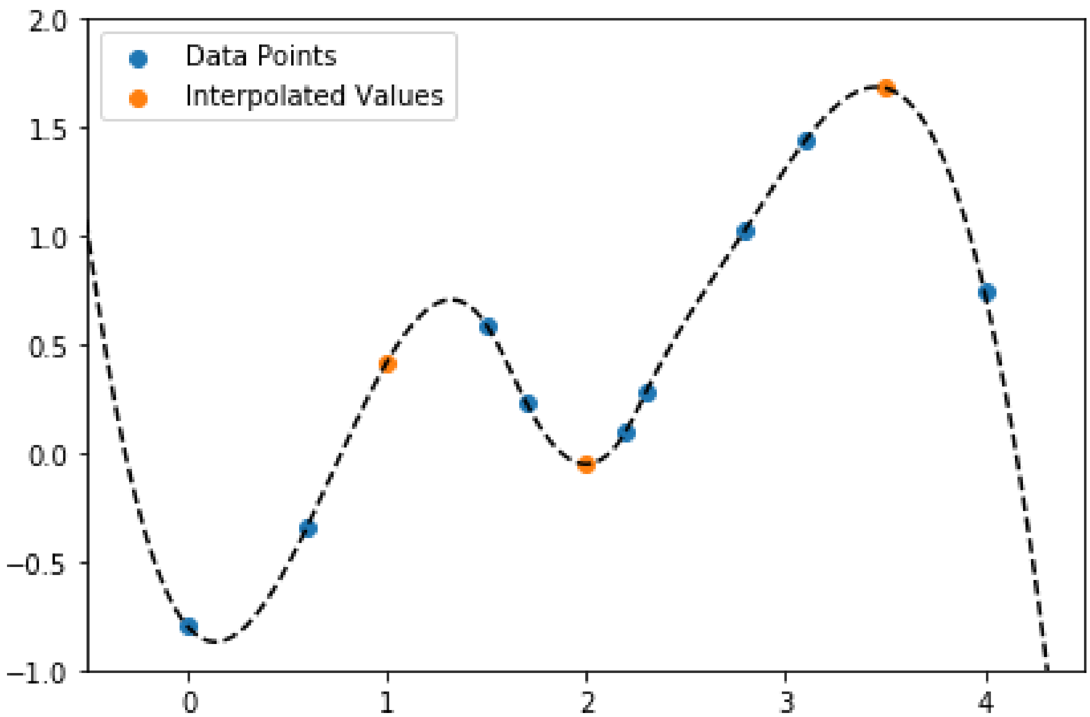


(e) B spline Interpolation

[15]:
```
result=interpolate.splrep(data_x,data_y,s=0)
x_range=np.arange(-3.,5.,0.01)

for x_i in interp_x:
    print ('for x={:.3f}, the expected value is y={:.3f}'.format(x_i,interpolate.
 ↪splev(x_i,result,der=0)))

plt.figure(figsize=(6,4))
plt.scatter(data_x,data_y,label='Data Points')
plt.scatter(interp_x,interpolate.
 ↪splev(interp_x,result,der=0),label='Interpolated Values')
```

```
plt.plot(x_range,interpolate.splev(x_range,result,der=0),c='k',ls='--')
plt.xlim(-0.5,4.5)
plt.ylim(-1.,2.)
plt.legend()
plt.tight_layout()
```

```
for x=1.000, the expected value is y=0.418
for x=2.000, the expected value is y=-0.051
for x=3.500, the expected value is y=1.682
```



4. $M_{BH} - \sigma_e$ Relation

[16]:
```
M,M_err,sigma,sigma_err=np.loadtxt('BlackHall.txt',unpack=True,usecols=[0,1,2,3])

def lin_func(p,x):
    return p[0]+p[1]*x
lin_model=Model(lin_func)
```

(a) without ΔM_{BH}, $\Delta\sigma_e$

[17]:
```
data_a=RealData(np.log10(sigma),np.log10(M))
odr=ODR(data_a,lin_model,beta0=[0.,1.0])

out=odr.run()
out.pprint()
p_a,perr_a,chi_a=out.beta,out.sd_beta,out.sum_square
```

```
Beta: [-3.95085978  5.36449855]
Beta Std Error: [1.18930851 0.53193191]
Beta Covariance: [[105.91343686 -47.27150679]
 [-47.27150679  21.18722599]]
Residual Variance: 0.01335481847952543
Inverse Condition #: 0.0027811945433923644
Reason(s) for Halting:
  Sum of squares convergence
```

(b) with ΔM_{BH}, $\Delta\sigma_e$

어떠한 측정값 x의 오차가 Δx라면, $f(x)$의 오차는 $\Delta f = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 (\Delta x)^2}$가 된다.
$f(x) = \log_{10} x$이므로, $\Delta f = \frac{\Delta x}{x \ln 10}$이다.

[18]:
```
data_b=RealData(np.log10(sigma),np.log10(M),\
                sx=sigma_err/(np.log(10)*sigma),sy=M_err/(np.log(10)*M))
odr=ODR(data_b,lin_model,beta0=[0.,1.0])

out=odr.run()
out.pprint()
p_b,perr_b,chi_b=out.beta,out.sd_beta,out.sum_square
```

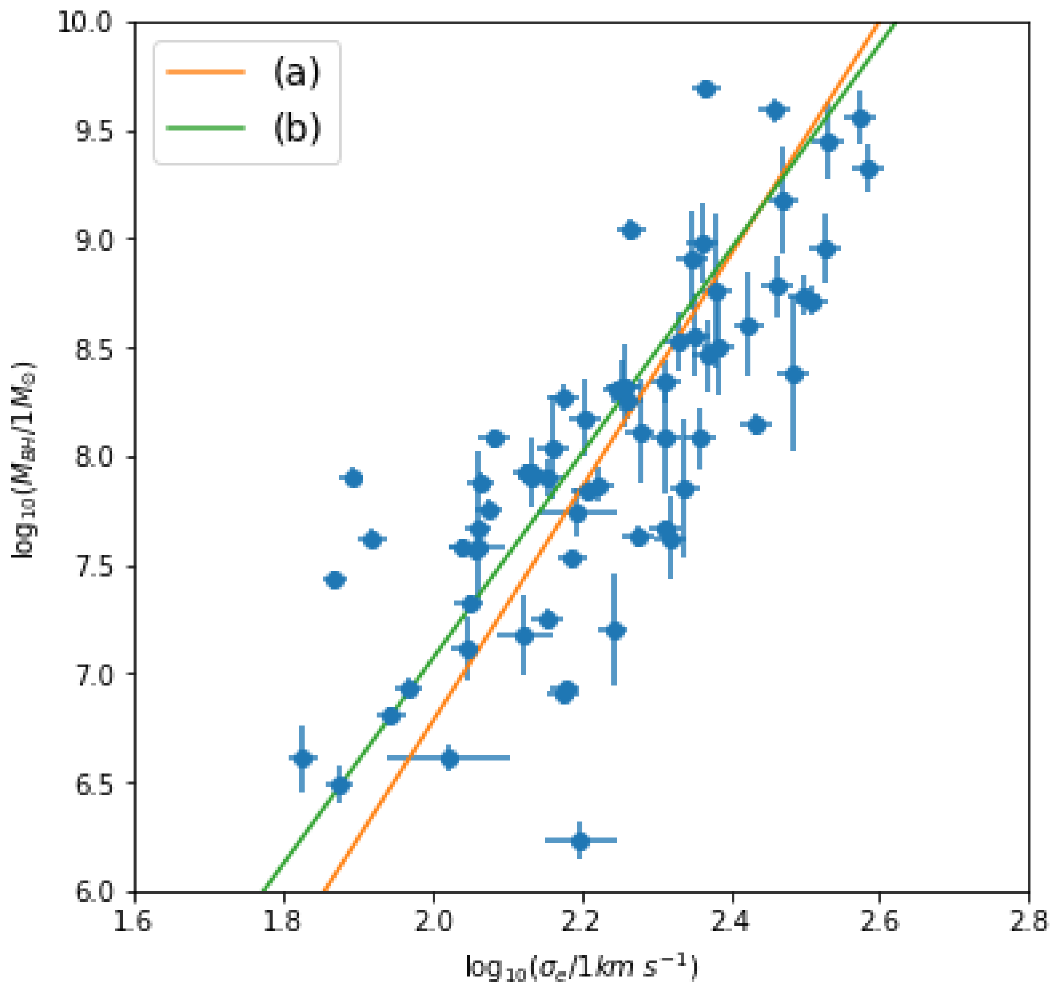
```
Beta: [-2.3397288   4.70367267]
Beta Std Error: [0.98627097 0.45017205]
Beta Covariance: [[ 0.04820329 -0.02193875]
 [-0.02193875  0.01004249]]
Residual Variance: 20.179753150752227
Inverse Condition #: 0.003314649510924227
Reason(s) for Halting:
  Sum of squares convergence
```

(c) Plot

[19]:
```
sigma_plot=np.linspace(1.5,3.0,100)

f=plt.figure(figsize=(6,6))
plt.errorbar(np.log10(sigma),np.log10(M),\
             M_err/(np.log(10)*M),sigma_err/(np.log(10)*sigma),\
```

```
                ls='none',marker='o')
plt.plot(sigma_plot,p_a[0]+sigma_plot*p_a[1],label='(a)')
plt.plot(sigma_plot,p_b[0]+sigma_plot*p_b[1],label='(b)')
plt.xlim(1.6,2.8)
plt.ylim(6.0,10.0)
plt.xlabel(r'$\log_{10}(\sigma_e/1km$ $s^{-1})$')
plt.ylabel(r'$\log_{10}(M_{BH}/1M_{\odot})$')
plt.legend(fontsize=14)
```

[19]: <matplotlib.legend.Legend at 0x1e56ccd6e48>



(d)

[20]:
```
data_d=RealData(np.log10(M),np.log10(sigma),\
                sx=M_err/(np.log(10)*M),sy=sigma_err/(np.log(10)*sigma))
odr=ODR(data_d,lin_model,beta0=[0.,1.0])

out=odr.run()
out.pprint()
p_d,perr_d,chi_d=out.beta,out.sd_beta,out.sum_square
```

```
Beta: [0.49738718 0.21260476]
Beta Std Error: [0.16225355 0.02034756]
Beta Covariance: [[ 1.30458554e-03 -1.62818514e-04]
 [-1.62818514e-04  2.05167564e-05]]
Residual Variance: 20.179753136850934
Inverse Condition #: 0.001212779254091
Reason(s) for Halting:
  Sum of squares convergence
```

(d)에서 구한 식, $\log(\sigma_e/1km\ s^{-1}) = c + d\log(M_{BH}/1M_\odot)$ 을 변형하면,

$$\log(M_{BH}/1M_\odot) = -c/d + (1/d)\log(\sigma_e/1km\ s^{-1}) \tag{5}$$
$$= a + b\log(\sigma_e/1km\ s^{-1}) \tag{6}$$

임을 알 수 있다. 이를 확인해본 결과는 아래와 같다.

[21]:
```
print ('a={:.3f}, -c/d={:.3f}'.format(p_b[0],-p_d[0]/p_d[1]))
print ('b={:.3f}, 1/d={:.3f}'.format(p_b[1],1./p_d[1]))
```

```
a=-2.340, -c/d=-2.339
b=4.704, 1/d=4.704
```

5. Gaussian & Lorentzian Fitting

[22]:
```
x,y=np.loadtxt('hw3p5.dat',unpack=True,usecols=[0,1])
```

(a) Gaussian Fitting

[23]:
```
def gauss_func(p,x):
    return p[0]+p[1]*np.exp(-0.5*pow((x-p[2])/p[3],2.))
gauss_model=Model(gauss_func)

data_a=RealData(x,y)
odr=ODR(data_a,gauss_model,beta0=[2.0,2.0,10.0,1.0]) # beta0의 값을 적절하게 지정

out=odr.run()
```

```
out.pprint()
p,perr,chi_a=out.beta,out.sd_beta,out.sum_square
```

```
Beta: [2.2459686  1.21002543 9.85323444 3.26830445]
Beta Std Error: [0.0322568  0.04023559 0.10167458 0.15083993]
Beta Covariance: [[ 2.65105822e-02 -2.05544514e-02  3.24591287e-05
-9.17721745e-02]
 [-2.05544514e-02  4.12474621e-02  6.12190876e-03  2.47633367e-02]
 [ 3.24591287e-05  6.12190876e-03  2.63391269e-01 -1.94031829e-02]
 [-9.17721745e-02  2.47633367e-02 -1.94031829e-02  5.79707922e-01]]
Residual Variance: 0.03924853139189
Inverse Condition #: 0.05272362027821306
Reason(s) for Halting:
  Sum of squares convergence
```

[24]:
```
print (p)
print (perr)
print (chi_a)
```

```
[2.2459686  1.21002543 9.85323444 3.26830445]
[0.0322568  0.04023559 0.10167458 0.15083993]
7.69271215281044
```

(b) Lorentzian Fitting

[25]:
```
def lorentz_func(q,x):
    return q[0]+q[1]/(q[2]+pow(x-q[3],2.))
lorentz_model=Model(lorentz_func)

data_b=RealData(x,y)
odr=ODR(data_b,lorentz_model,beta0=[2.0,2.0,15.0,1.0]) # beta0의 값을 적절하게 지정

out=odr.run()
out.pprint()
q,qerr,chi_b=out.beta,out.sd_beta,out.sum_square
```

```
Beta: [ 1.98482168 28.60772792 19.05848266  9.81597799]
Beta Std Error: [0.05842486 4.76748905 2.74750963 0.10289067]
Beta Covariance: [[ 8.68796022e-02 -6.64954719e+00 -3.55681025e+00
-2.23424691e-03]
 [-6.64954719e+00  5.78497979e+02  3.26837672e+02  1.17303770e-02]
 [-3.55681025e+00  3.26837672e+02  1.92132522e+02 -8.16618950e-02]
 [-2.23424691e-03  1.17303770e-02 -8.16618950e-02  2.69447671e-01]]
Residual Variance: 0.039289595995299284
Inverse Condition #: 0.002893204496777257
Reason(s) for Halting:
  Sum of squares convergence
```

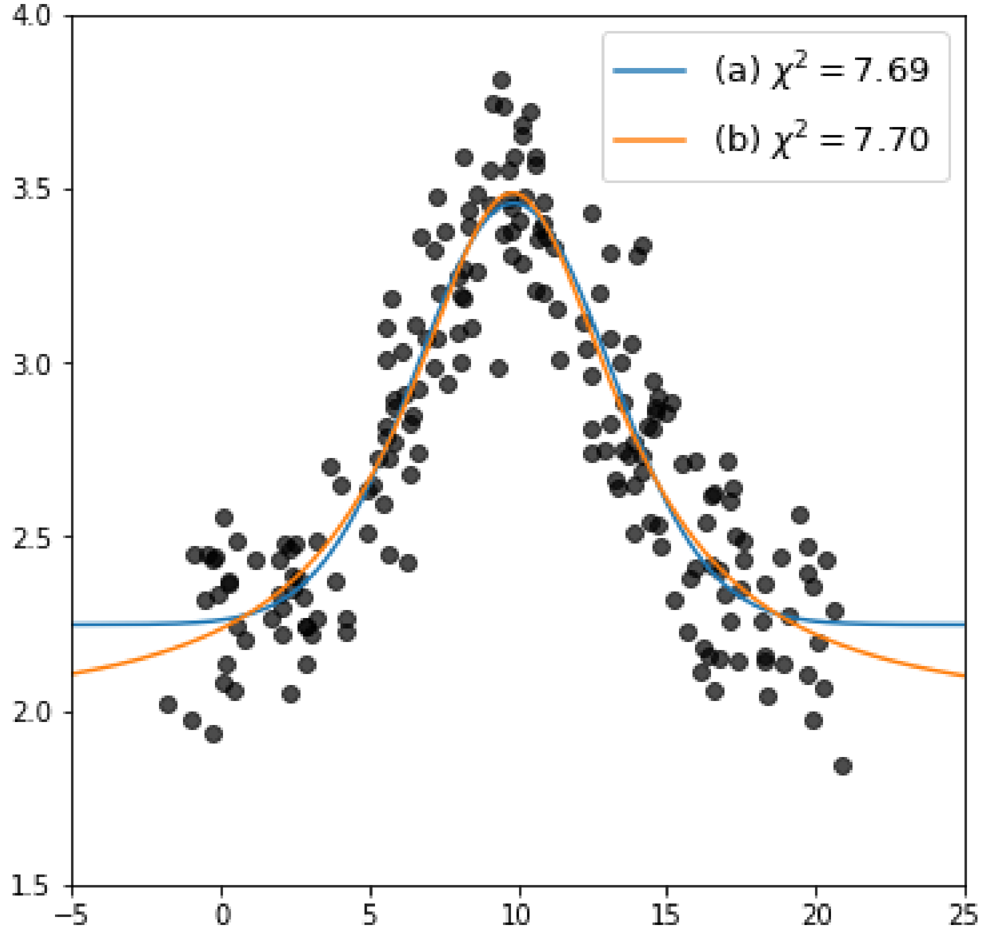
[26]:
```
print (q)
print (qerr)
print (chi_b)
```

```
[ 1.98482168 28.60772792 19.05848266  9.81597799]
[0.05842486 4.76748905 2.74750963 0.10289067]
7.7007608150786595
```

(c) Results

[27]:
```
x_plt=np.linspace(-5.,25.,1000)
def gauss_plt(x):
    return p[0]+p[1]*np.exp(-0.5*pow((x-p[2])/p[3],2.))
def lorentz_plt(x):
    return q[0]+q[1]/(q[2]+pow(x-q[3],2.))

plt.figure(figsize=(6,6))
plt.scatter(x,y,c='k',alpha=0.7)
plt.plot(x_plt,gauss_plt(x_plt),label=r'(a) $\chi^2={:.2f}$'.format(chi_a))
plt.plot(x_plt,lorentz_plt(x_plt),label='(b) $\chi^2={:.2f}$'.format(chi_b))
plt.xlim(-5.,25.)
plt.ylim(1.5,4.0)
plt.legend(fontsize=13)
```

[27]:
```
<matplotlib.legend.Legend at 0x1e56c622630>
```



Gaussian이 Lorentzian 보다 χ^2이 미세하게 적다.

1) 그러므로 Gaussian이 더 좋은 Fitting이라고 결론 내릴 수 있다.

2) 하지만 그 차이가 매우 적기때문에, 이를 바탕으로 두 함수중 중 어느 것이 더 좋은 Fitting 결과를 주는지 판단을 내리기는 어렵다.