Problem Set 1

Aaron Desrochers

Due Friday September 16th at 11:59pm

Problem 1:

Problem 1: We saw in class how Taylor series/roundoff errors fight against each other when deciding how big a step size to use when calculating numerical derivatives. If we allow ourselves to evaluate our function f at four points ($x \pm \delta$ and $x \pm 2\delta$),
a) what should our estimate of the first derivative at x be? Rather than doing a complicated fit, I suggest thinking about how to combine the derivative from $x \pm \delta$ with the derivative from $x \pm 2\delta$ to cancel the next term in the Taylor series.
b) Now that you have your operator for the derivative, what should δ be in terms of the machine precision and various properties of the function? Show for $f(x) = \exp(x)$ and $f(x) = \exp(0.01x)$ that your estimate of the optimal δ is at least roughly correct.

I forgot that I skipped over this question initially, and I'm only realizing now, right before I need to hand in the assignment. Oops (:

Problem 2:

Problem 2: Write a numerical differentiator with prototype

```
def ndiff(fun,x,full=False):
```

where fun is a function and x is a value. If full is set to False, ndiff should return the numerical derivative at x. If full is True, it should return the derivative, dx, and an estimate of the error on the derivative. I suggest you use the centered derivative

$$f' \simeq \frac{f(x+dx) - f(x-dx)}{2\mathrm{d}x}$$

Your routine should estimate the optimal dx then use that in calculating the derivative. If you're feeling ambitious, write your code so that x can be an array, not just a single number. If you do that, you may actually wish to save your code as you might use it in the future.

In [20]:
```
import numpy as np
#from scipy.misc import derivative


def ndiff(fun, x, full=False):
    #Find dx
    Em = 1e-16 #machine error for double precision machines
    Ef = Em #for simple functions, the fractional accuracy is
            #comparable to the machine accuracy
    dx = Ef**(1/3)*x

    #Calculate the symmetric derivative

    df = (fun(x+dx)-fun(x-dx))/(2*dx)

    #estimate error

    error = Ef**(2/3)

    if full==False:
        return df

    elif full==True:
        return [df, dx, error]

    else:
        print("full must be either True or False")
```

Clarification: dx: The equation for dx ($\epsilon_f^{\frac{1}{3}} x$) comes from the fact that we want to find a dx that minimizes the sum of the truncation error and the roundoff error. We have equations for the truncation error: $e_t \approx dx^2 f'''$, and roundoff error: $e_r \approx \epsilon_f |\frac{f}{dx}|$, so by minimizing the sum of these two errors, we come to the equation for dx: $dx \approx (\frac{\epsilon_f f}{f'''})^{\frac{1}{3}}$, which is approximately equal to $\epsilon_f^{\frac{1}{3}} x$.

error: To calculate the error, we again need the truncation errorr and the roundoff error. The equation for the error is written as: $\frac{(e_r+e_t)}{|f'|}$, which then, by plugging in values for e_r and e_t becomes $(\epsilon_f)^{\frac{2}{3}} f^{\frac{2}{3}} (f''')^{\frac{1}{3}}/f'$, which approximates to $\epsilon_f^{\frac{2}{3}}$.

Problem 3:

Problem 3: Lakeshore 670 diodes (successors to the venerable Lakeshore 470) are temperature-sensitive diodes used for a range of cryogenic temperature measurements. They are fed with a constant 10 μA current, and the voltage is read out. Lakeshore provides a chart that converts voltage to temperature, available at https://www.lakeshore.com/products/categories/specification/temperature-products/cryogenic-temperature-sensors/dt-670-silicon-diodes, or you can look at the text file I've helpfully copied and pasted (lakeshore.txt). Write a routine that will take an arbitrary voltage and interpolate to return a temperature. You should also make some sort of quantitative (but possibly rough) estimate of the error in your interpolation as well (this is a common situation where you have been presented with data and have to figure out *some* idea of how to get error estimates).

Your prototype should be:

```
def lakeshore(V,data):
```

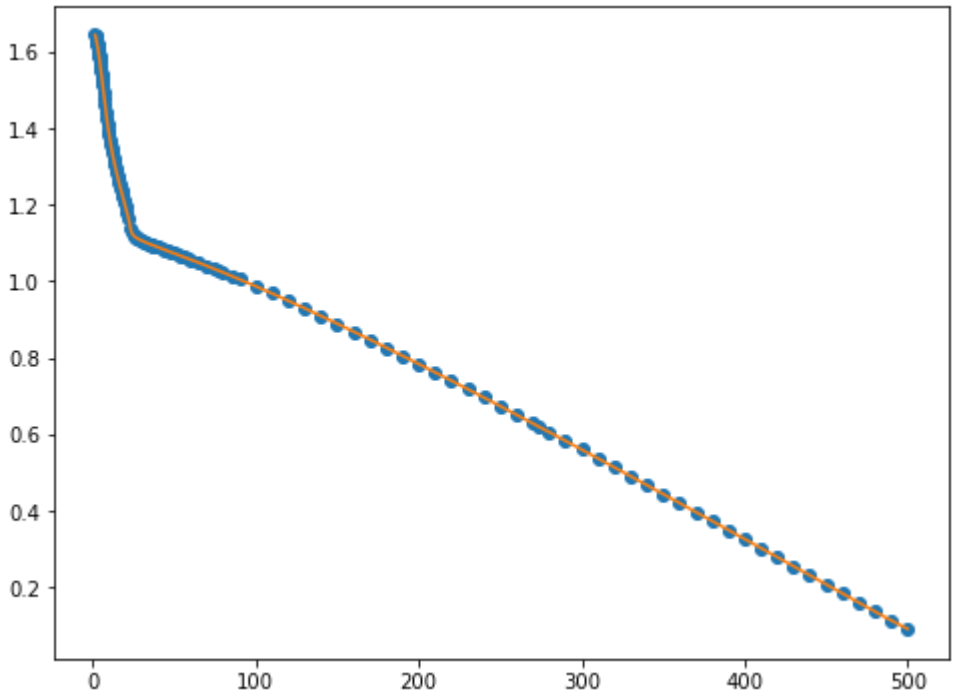
where data is the output of:

```
dat=np.loadtxt('lakeshore.txt')
```

The columns of lakeshore.txt are 1) the temperature, 2) the voltage across the diode at that temperature (which is what your experiment will actually measure), and 3) dV/dT. You do not need to use the third column, but you may if you wish.

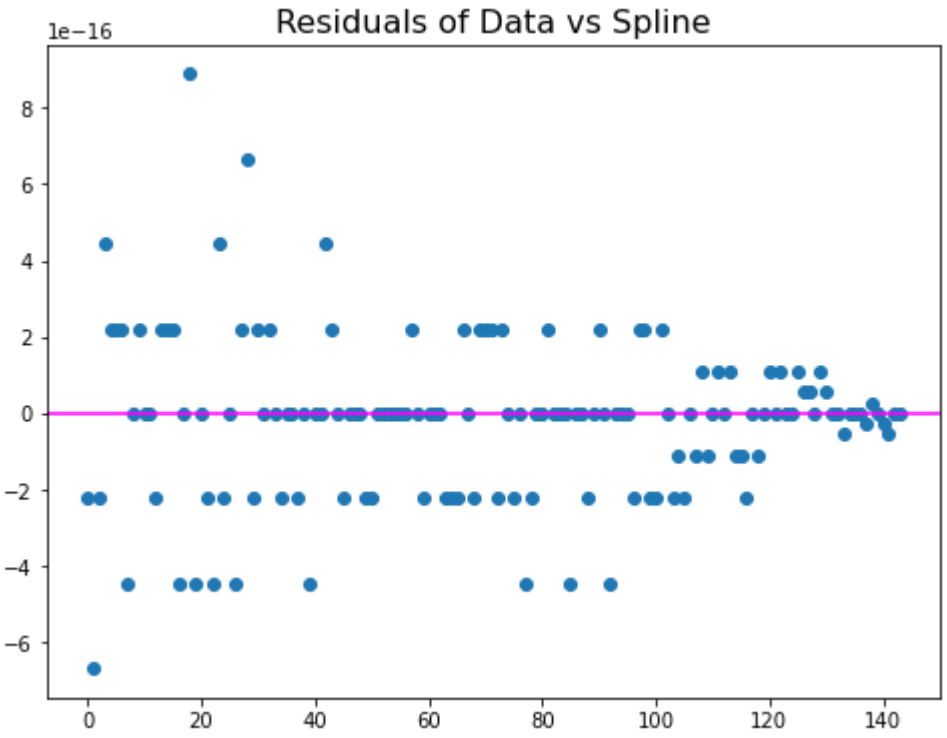
Your code should support V being either a number or array, and it should return the interpolated temperature, and your estimated uncertainty on the temperature.

In [64]:
```
#Looking at the data
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate

data = np.loadtxt('lakeshore.txt')

temp = data[:,0]
voltage = data[:,1]

x = np.linspace(min(temp), max(temp), 1000)

spline = interpolate.splrep(temp, voltage)
y = interpolate.splev(x, spline)

plt.figure(figsize=(8,6))
plt.plot(temp, voltage, 'o')
plt.plot(x, y)
plt.show()
```



In [21]:
```
#coding the function
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate

def lakeshore(V, data):
    #separate temp and voltage data
    t = data[:,0]
    v = data[:,1]
    #create the spline from our data
    spline = interpolate.splrep(t, v)
    #find the temp of the input voltage
    temp = interpolate.splev(V, spline)

    return temp
```

In [3]:
```
#Roughly estimating the data
data = np.loadtxt('lakeshore.txt')

temp = data[:,0]
voltage = data[:,1]

x = np.linspace(min(temp), max(temp), 1000)

spline = interpolate.splrep(temp, voltage)
y = interpolate.splev(x, spline)

err = np.zeros(len(temp))

for i in range(len(temp)):
    err[i]=voltage[i]-interpolate.splev(temp[i], spline)

plt.figure(figsize=(8,6))
plt.title('Residuals of Data vs Spline', fontsize=16)
plt.plot(err, 'o')
plt.axhline(0, color='magenta')
plt.show()

print(np.std(err))
```




```
2.1316316525753148e-16
```

To estimate the error, the residuals of the cubic spline with the data points were computed and can be seen to be on the order of 10^{-16}, which is on the same order as the limit of our computer. Another thing to consider is that the measured values of voltage are only accurate up to 10^{-6}, and the overall error can't be lower than the error on the measurement itself. Taking this into account, the error on the interpolation is around 10^{-6}.

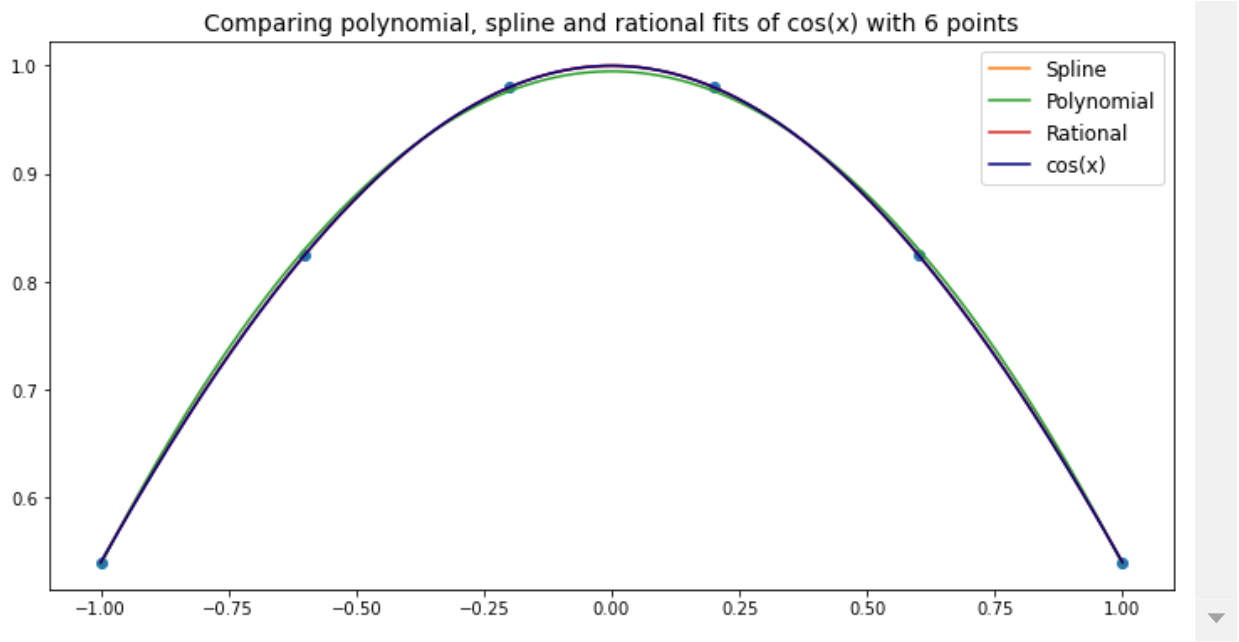
Problem 4:

Problem 4: Take $\cos(x)$ between $-\pi/2$ and $\pi/2$. Compare the accuracy of polynomial, cubic spline, and rational function interpolation given some modest number of points, but for fairness each method should use the same points. Now try using a Lorentzian $1/(1+x^2)$ between -1 and 1.

What should the error be for the Lorentzian from the rational function fit? Does what you got agree with your expectations when the order is higher (say n=4, m=5)? What happens if you switch from np.linalg.inv to np.linalg.pinv (which tries to deal with singular matrices)? Can you understand what has happened by looking at p and q? As a hint, think about why we had to fix the constant term in the denominator, and how that might generalize.

Cosine

In [23]:
```
#Function
n=3 #numerator & denominator order for rational fit
m=2

func=np.cos

x = np.linspace(-1,1,n+m+1)
y = func(x)

xx = np.linspace(-1, 1, 1001)

#Poly
y_poly=np.zeros(len(xx))

for i, myx in enumerate(xx):
    pp=np.polyfit(x, y, 3)
    y_poly[i]=np.polyval(pp,myx)

#Spline
spline = interpolate.splrep(x, y)
y_spline = interpolate.splev(xx, spline)

#Rational
pcols=[x**k for k in range(n+1)]
pmat=np.vstack(pcols)

qcols=[-x**k*y for k in range(1,m+1)]
qmat=np.vstack(qcols)
mat=np.hstack([pmat.T,qmat.T])
coeffs=np.linalg.inv(mat)@y

p=0
for i in range(n+1):
    p=p+coeffs[i]*xx**i
qq=1
for i in range(m):
    qq=qq+coeffs[n+1+i]*xx**(i+1)

y_rat=p/qq


plt.figure(figsize=(12,6))
plt.title('Comparing polynomial, spline and rational fits of cos(x) with {} point
plt.plot(x,y, 'o')
plt.plot(xx, y_spline, label='Spline')
plt.plot(xx, y_poly, label='Polynomial')
plt.plot(xx, y_rat, label='Rational')
plt.plot(xx, func(xx), label='cos(x)', color='navy')
plt.legend(fontsize=12)
plt.show()

print('Error in polynomial fit: ',np.std(y_poly-func(xx)))
print('Error in cubic spline: ',np.std(y_spline-func(xx)))
print('Error in rational fit: ',np.std(y_rat-func(xx)))
```

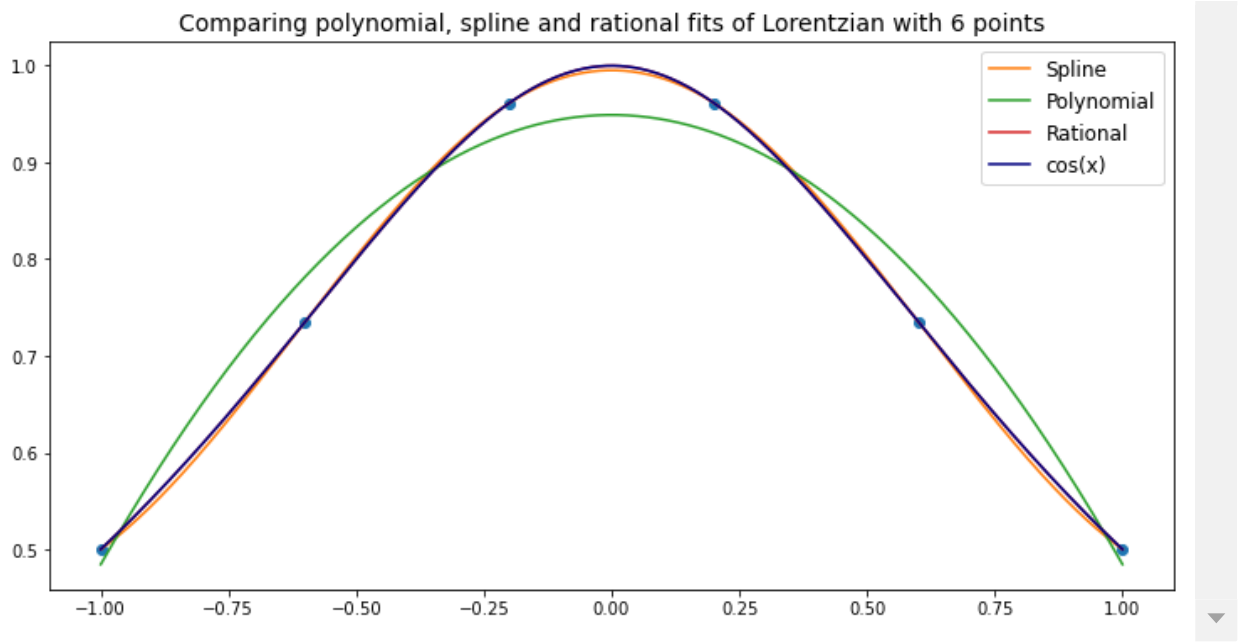



```
Error in polynomial fit:  0.004047370486308816
Error in cubic spline:  0.00027611698951646776
Error in rational fit:  5.86253595415372e-05
```

As we can see from the error on each fit, the accuracy increases as we go from a polynomial fit to a cubic spline, then increases again as we go to a rational fit.

Lorentz

In [13]:
```
#Function
n=3 #numerator & denominator order for rational fit
m=2

def func(x):
    return 1/(1+x**2)

x = np.linspace(-1,1,n+m+1)
y = func(x)

xx = np.linspace(-1, 1, 1001)

#Poly
y_poly=np.zeros(len(xx))

for i, myx in enumerate(xx):
    pp=np.polyfit(x, y, 3)
    y_poly[i]=np.polyval(pp,myx)

#Spline
spline = interpolate.splrep(x, y)
y_spline = interpolate.splev(xx, spline)

#Rational
pcols=[x**k for k in range(n+1)]
pmat=np.vstack(pcols)

qcols=[-x**k*y for k in range(1,m+1)]
qmat=np.vstack(qcols)
mat=np.hstack([pmat.T,qmat.T])
coeffs=np.linalg.inv(mat)@y

p=0
for i in range(n+1):
    p=p+coeffs[i]*xx**i
qq=1
for i in range(m):
    qq=qq+coeffs[n+1+i]*xx**(i+1)

y_rat=p/qq


plt.figure(figsize=(12,6))
plt.title('Comparing polynomial, spline and rational fits of Lorentzian with {} p
plt.plot(x,y, 'o')
plt.plot(xx, y_spline, label='Spline')
plt.plot(xx, y_poly, label='Polynomial')
plt.plot(xx, y_rat, label='Rational')
plt.plot(xx, func(xx), label='cos(x)', color='navy')
plt.legend(fontsize=12)
plt.show()

print('Error in polynomial fit: ',np.std(y_poly-func(xx)))
print('Error in cubic spline: ',np.std(y_spline-func(xx)))
print('Error in rational fit: ',np.std(y_rat-func(xx)))
```

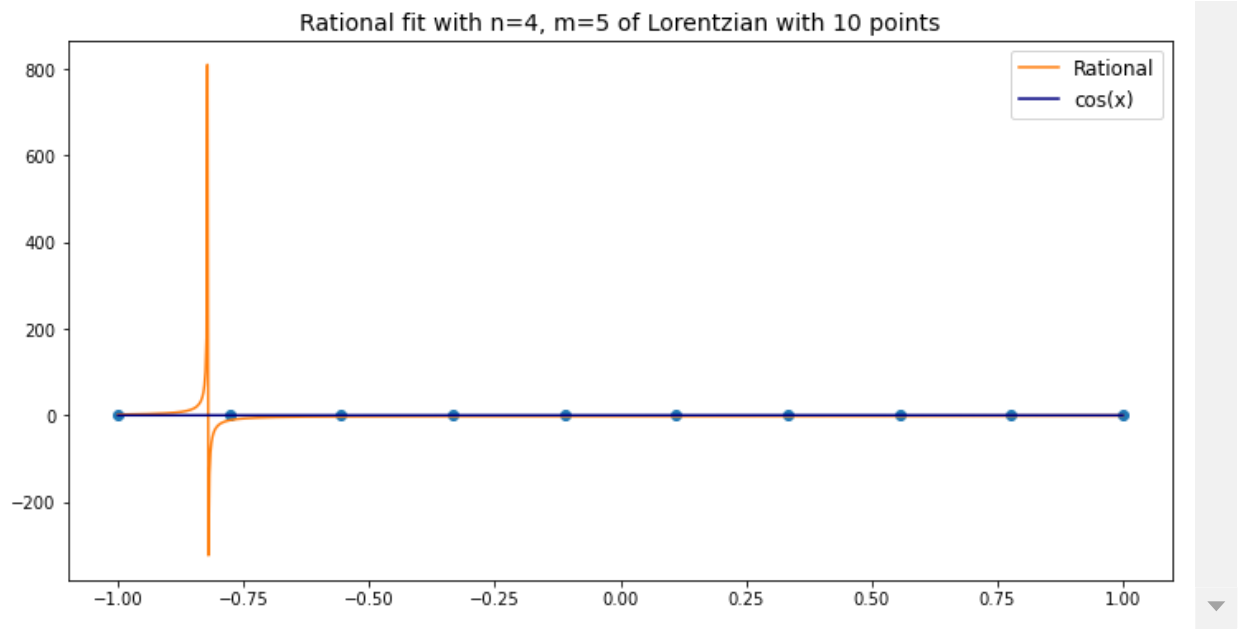



```
Error in polynomial fit:  0.034338085497428145
Error in cubic spline:  0.0038300363626882373
Error in rational fit:  7.780468787861449e-16
```

The error we should expect to see on the Lorentzian with the rational function fit is 0, since the Lorentzian itself IS a rational function. What we actually get is error on the order of 10^{-16}, which is caused by the limits of the computer.

Trying with new numerator/denominator

In [15]:
```
#Function
n=4 #numerator & denominator order for rational fit
m=5

def func(x):
    return 1/(1+x**2)

x = np.linspace(-1,1,n+m+1)
y = func(x)

xx = np.linspace(-1, 1, 1001)

#Rational
pcols=[x**k for k in range(n+1)]
pmat=np.vstack(pcols)

qcols=[-x**k*y for k in range(1,m+1)]
qmat=np.vstack(qcols)
mat=np.hstack([pmat.T,qmat.T])
coeffs=np.linalg.inv(mat)@y

p=0
for i in range(n+1):
    p=p+coeffs[i]*xx**i
qq=1
for i in range(m):
    qq=qq+coeffs[n+1+i]*xx**(i+1)

y_rat=p/qq


plt.figure(figsize=(12,6))
plt.title('Rational fit with n=4, m=5 of Lorentzian with {} points'.format(n+m+1)
plt.plot(x,y, 'o')
plt.plot(xx, y_rat, label='Rational')
plt.plot(xx, func(xx), label='cos(x)', color='navy')
plt.legend(fontsize=12)
plt.show()

print('Error in rational fit: ',np.std(y_rat-func(xx)))
```

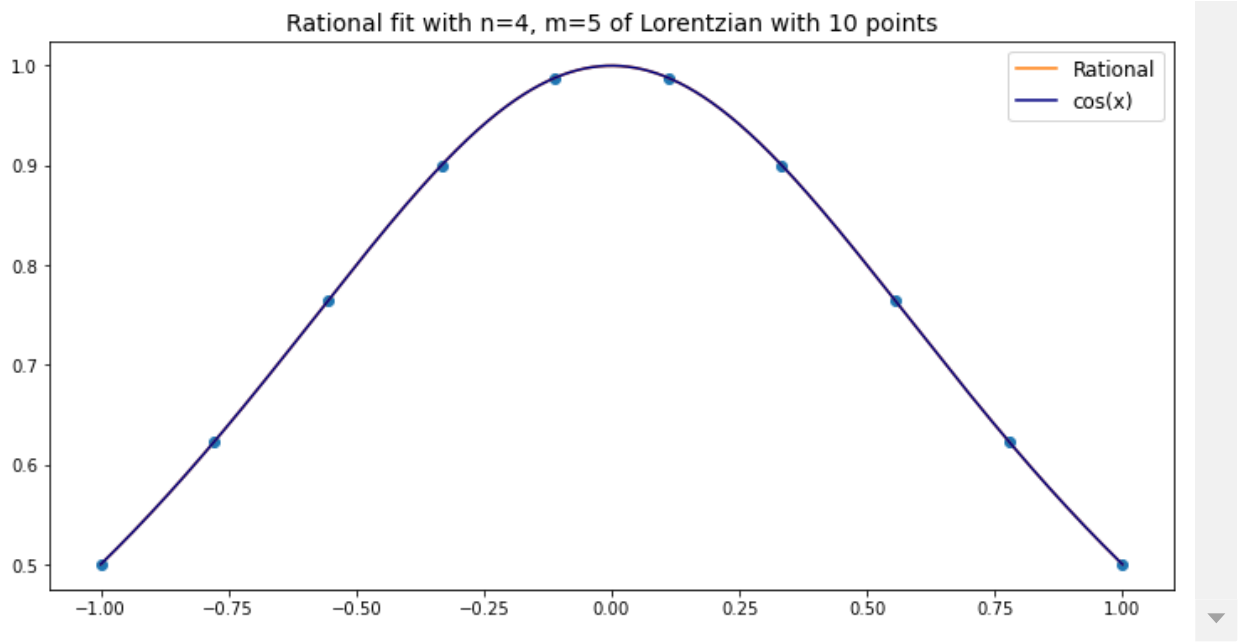



```
Error in rational fit:  29.365272059433355
```

Now that we changed the order of our numerator and denominator, the rational fit is messing up. The fit no longer agrees with our expectations.

Changing to pinv

In [16]:
```
#Function
n=4 #numerator & denominator order for rational fit
m=5

def func(x):
    return 1/(1+x**2)

x = np.linspace(-1,1,n+m+1)
y = func(x)

xx = np.linspace(-1, 1, 1001)

#Rational
pcols=[x**k for k in range(n+1)]
pmat=np.vstack(pcols)

qcols=[-x**k*y for k in range(1,m+1)]
qmat=np.vstack(qcols)
mat=np.hstack([pmat.T,qmat.T])
coeffs=np.linalg.pinv(mat)@y

p=0
for i in range(n+1):
    p=p+coeffs[i]*xx**i
qq=1
for i in range(m):
    qq=qq+coeffs[n+1+i]*xx**(i+1)

y_rat=p/qq


plt.figure(figsize=(12,6))
plt.title('Rational fit with n=4, m=5 of Lorentzian with {} points'.format(n+m+1)
plt.plot(x,y, 'o')
plt.plot(xx, y_rat, label='Rational')
plt.plot(xx, func(xx), label='cos(x)', color='navy')
plt.legend(fontsize=12)
plt.show()

print('Error in rational fit: ',np.std(y_rat-func(xx)))
```




```
Error in rational fit:  3.737098796860195e-16
```

When we switch from np.linalg.inv to np.linalg.pinv, even with our numerator and denominator set at 4 & 5 respectively, our rational fit works! This is because when we try to calculate our coefficients, our program will correctly predict that the 3rd, 4th and 5th order terms in our denominator should be equal to 0, since we are fitting to a Lorentzian ($\frac{1}{1+x^2}$). These 0 terms in the coefficiennt matrix then cause problems when we take the inverse of said matrix with using np.linalg.inv. Luckily though, we can instead take the "pseudo-inverse" of the matrix by using np.linalg.pinv, which lets us bypass the problems caused by the 0s and therefore giving us a proper rationanl fit.