In [4]:
```
import numpy as np
from matplotlib import pyplot as plt
from scipy import interpolate as interp
from scipy import stats
import warnings; warnings.simplefilter('ignore')
```

Note that Loic Miara and me helped each other on this assignment which can explain the ressemblance between our assignments.

PROBLEM ONE

a)

The first terms of the Taylor expansion at each 4 points are:

$f(x+\delta) = f(x) + f^1(x)\delta + \frac{1}{2}f^2(x)\delta^2 + \frac{1}{6}f^3(x)\delta^3 + \frac{1}{24}f^4(x)\delta^4 + \frac{1}{120}f^5(x)\delta^5 \; + \; ...$

$f(x-\delta) = f(x) - f^1(x)\delta + \frac{1}{2}f^2(x)\delta^2 - \frac{1}{6}f^3(x)\delta^3 + \frac{1}{24}f^4(x)\delta^4 - \frac{1}{120}f^5(x)\delta^5 \; + \; ...$

$f(x+2\delta) = f(x) + f^1(x)\delta + 2f^2(x)\delta^2 + \frac{8}{6}f^3(x)\delta^3 + \frac{16}{24}f^4(x)\delta^4 + \frac{32}{120}f^5(x)\delta^5 \; + \; ...$

$f(x-2\delta) = f(x) - f^1(x)\delta + 2f^2(x)\delta^2 - \frac{8}{6}f^3(x)\delta^3 + \frac{16}{24}f^4(x)\delta^4 - \frac{32}{120}f^5(x)\delta^5 \; + \; ...$

Now, by substracting $f(x+\delta)$ and $f(x-\delta)$, the even terms cancel out such that,

$\frac{f(x+\delta)-f(x-\delta)}{2\delta} \approx f^1(x) + \frac{1}{6}f^3(x)\delta^2 + \frac{1}{120}f^5(x)\delta^4 \; + \; ...$ (1)

Similarly, by substracting $f(x+2\delta)$ and $f(x-2\delta)$, the even terms cancel out such that,

$\frac{f(x+2\delta)-f(x-2\delta)}{4\delta} \approx 3f^1(x) + \frac{2}{3}f^3(x)\delta^2 + \frac{32}{120}f^5(x)\delta^4 \; + \; ...$ (2)

We can cancel the third derivative term of the Taylor series by substracting 4*(1) from (2), such that

$\frac{f(x+2\delta)-f(x-2\delta)}{4\delta} - 4\frac{f(x+\delta)-f(x-\delta)}{2\delta} = -3f^1(x) + \frac{7}{30}f^5(x)\delta^4 \; + \; ...$

With a truncation error $\delta^4 f^5(x)$, we obtain the following numerical approximation for $f'(x)$,

$f'(x) \approx \frac{8[f(x+\delta)-f(x-\delta)]-[f(x+2\delta)-f(x-2\delta)]}{12\delta}$

b)

The total error is the sum of the round-off and truncation error, which is $\approx \frac{f\epsilon}{\delta} + \delta^4 f^5(x)$, where ϵ is the machine precision (here 10^{-16}).

Set it equal to zero and differentiate with respect to δ to minimize the error:

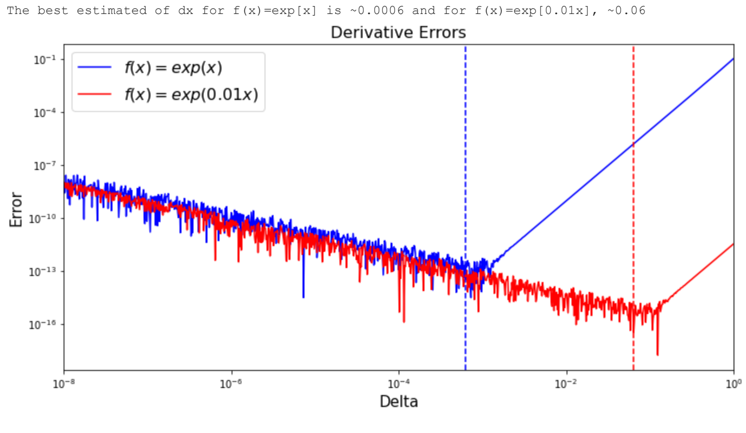
$\frac{f\epsilon}{\delta^2} + 4\delta^3 f^5(x) = 0 \iff -f\epsilon + 4\delta^5 f^5(x) = 0$
$\iff \delta \approx (\epsilon\frac{f}{f^5})^{1/5}$ (1)

Now, $\frac{f}{f^5}$ for $f(x) = e^x$ is 1 for all x and $\frac{f}{f^5}$ for $f(x) = e^{0.01x}$ is 10^{10} for all x. Therefore, plugging these and ϵ in (1) yields

$\delta_{e^x} \approx 10^{-3}$ and $\delta_{e^{0.01x}} \approx 10^{-1}$.

In [5]:
```
#This code is inspired from John's "num_derivs_clean.py" code.

logdx=np.linspace(-8,0,1001)
dx=10**logdx #dx's range from 1e-8 to 1

fun=np.exp
x0=1 #the function is evaluated at x=1

#The four points where the function is evaluated
y1=fun(x0+dx)
y2=fun(x0-dx)
y3=fun(x0+2*dx)
y4=fun(x0-2*dx)

#Similarly, the four points evaluated for f(x) = exp[0.01x]
y11=fun((x0+dx)/100)
y22=fun((x0-dx)/100)
y33=fun((x0+2*dx)/100)
y44=fun((x0-2*dx)/100)

#The returned numerical derivatives
d1=1/(12*dx)*(8*(y1-y2)-(y3-y4))      #for f(x) = exp[x]
d2=1/(12*dx)*(8*(y11-y22)-(y33-y44))  #for f(x) = exp[0.01x]

#Compute the best dx estimation
def est_dx(ratio,n):
    return (1e-16*ratio)**n

n=1/5
ratio1=1
ratio2=1e10

dx1=est_dx(ratio1,n)
dx2=est_dx(ratio2,n)
print('The best estimated of dx for f(x)=exp[x] is ~{} \
and for f(x)=exp[0.01x], ~{}'.format(round(dx1,4),round(dx2,2)))

#Plot the Figure
fig = plt.subplots(1,figsize=(12,6))
ax=plt.gca()
plt.loglog(dx,np.abs(d1-np.exp(x0)),label="$f(x) = exp(x)$",color = 'b')
plt.loglog(dx,np.abs(d2-1/100*np.exp(x0/100)),label="$f(x) = exp(0.01x)$",color = 'r')
plt.title('Derivative Errors',fontsize=16)
plt.xlabel('Delta',fontsize=15)
plt.ylabel('Error',fontsize=15)
plt.legend(loc=2, prop={'size': 16})
plt.axvline(x = dx1, linestyle='--',color = 'b')
plt.axvline(x = dx2, linestyle='--',color = 'r')
ax.set_xlim(1e-8,1)
plt.show()
```

```
The best estimated of dx for f(x)=exp[x] is ~0.0006 and for f(x)=exp[0.01x], ~0.06
```

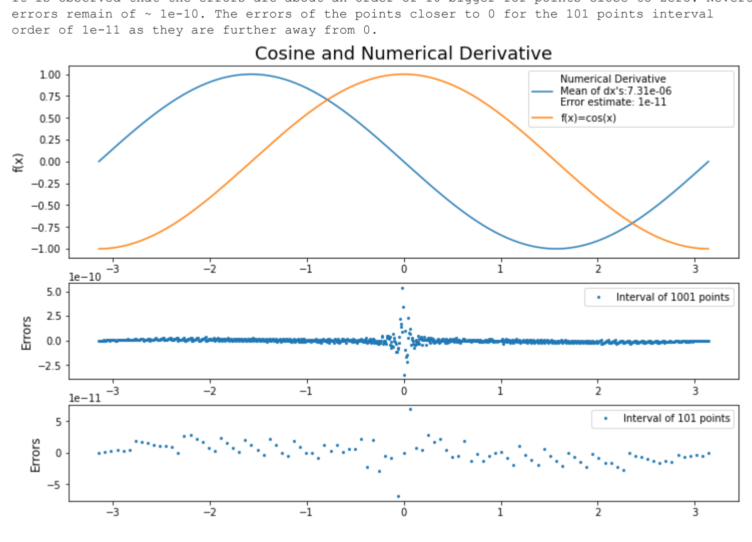


PROBLEM 2

In [6]:
```
#This code is inspired from John's "num_derivs_clean.py" code as well.

def ndiff(fun,x,Full=False,showPrints=True):
    try: #This try-except block is to test if the argument x is an array
        if len(x)!=0:
            #deletes the 0 element if present in the array since it could yield craziness
            if 0 in x:
                ind=np.where(x==0)
                x = np.delete(x,ind)
                print('The element 0 at index {} has been removed from the array'.format(ind[0]))
                #notifies that 0 is removed
    except:
        if x==0:
            print('Error: The testing point cannot be zero') #Test case because this method cannot use point at
            return None
    dx=1e-16**(1/3)*np.abs(x)
    #this is the formula to estimate the optimal dx (eq. 5.7.8 p.187 in Numerical Reciped in C, 2nd edition)
    #It is mentioned that when no information is given on the characteristic scale (i.e. (f/f^3)^(1/3)),
    #it is often assumed to be approximated by x (except at/near 0)

    deriv=(fun(x+dx)-fun(x-dx))/(2*dx) #centered derivative using two points

    if Full:
        fract_err=1e-16**(2/3) #Error estimation given in eq. 5.7.9 in Numerical Reciped in C, 2nd edition
        if showPrints:
            print("Derivative f'(x) = {}; dx={}; Error estimate={} ".format(deriv,dx,fract_err))
        return deriv,dx,fract_err
    else:
        if showPrints:
            print("Derivative f'(x)={}".format(deriv))
        return deriv
```

In [7]:
```
#Testing the function

#Test case if a 0 is entered as the testing point
print('Test with exponential function, testing point=0:')
deriv = ndiff(np.exp,0)
print('\n')

#Test case if 0 is in the array of the testing points
print('Test with exponential function, testing points contain a zero:')
deriv = ndiff(np.exp,np.linspace(0,5,11))
print('\n')

#Cosine
print('Test with cosine function:')
xx = np.linspace(-np.pi,np.pi,1001)
xx2 = np.linspace(-np.pi,np.pi,101)
print('Interval of 1001 points:')
deriv,dx,error = ndiff(fun=np.cos,x=xx,Full=True,showPrints=False)
x_use = np.delete(xx,500)
y_true = -np.sin(x_use)
print('Interval of 101 points:\n--')
deriv2 = ndiff(fun=np.cos,x=xx2,showPrints=False)
y_true2 = -np.sin(xx2)

fig,axs = plt.subplots(3,figsize=(12,8),gridspec_kw={'height_ratios': [2,1,1]})
axs[0].set_title('Cosine and Numerical Derivative',fontsize=18)
axs[0].plot(x_use,deriv,label="Numerical Derivative\nMean of dx's:{}\nError estimate: {}\
                  ".format('{:0.2e}'.format(np.mean(dx)),10**round(np.log10(error))))
axs[0].plot(xx,np.cos(xx),label='f(x)=cos(x)')
axs[0].set_ylabel('f(x)',fontsize=12)
axs[0].legend()
axs[1].plot(x_use,y_true-deriv,'o',markersize=2,label='Interval of 1001 points')
axs[1].set_ylabel('Errors',fontsize=12)
axs[1].legend()
axs[2].plot(xx2,y_true2-deriv2,'o',markersize=2,label='Interval of 101 points')
axs[2].set_ylabel('Errors',fontsize=12)
axs[2].legend()

print('It is observed that the errors are about an order of 10 bigger for points close to zero. Nevertheless,\
      the errors remain of ~ 1e-10. The errors of the points closer to 0 for the 101 points interval \
      remain in the order of 1e-11 as they are further away from 0.')
```

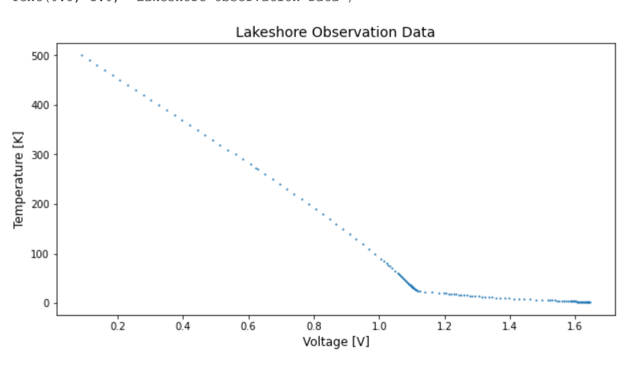
```
Test with exponential function, testing point=0:
Error: The testing point cannot be zero


Test with exponential function, testing points contain a zero:
The element 0 at index [0] has been removed from the array
Derivative f'(x)=[  1.64872127   2.71828183   4.48168907   7.3890561   12.18249396
  20.08553692  33.11545196  54.59815004  90.01713131 148.41315912]


Test with cosine function:
Interval of 1001 points:
The element 0 at index [500] has been removed from the array
Interval of 101 points:
--
It is observed that the errors are about an order of 10 bigger for points close to zero. Nevertheless,      the errors remain of ~ 1e-10. The errors of the points closer to 0 for the 101 points interval       remain in the order of 1e-11 as they are further away from 0.
```



PROBLEM 3

In [8]:
```
#Download the data
data = np.loadtxt("lakeshore.txt")

#data is flipped because scipy needs it to be ordered
voltages = np.flip(data[:,1])
temperatures = np.flip(data[:,0])

fig=plt.figure(figsize=(10,5))
plt.plot(voltages,temperatures,'o',markersize=1,)
plt.xlabel('Voltage [V]',fontsize=12)
plt.ylabel('Temperature [K]',fontsize=12)
plt.title('Lakeshore Observation Data',fontsize=14)
```

Out[8]:
```
Text(0.5, 1.0, 'Lakeshore Observation Data')
```



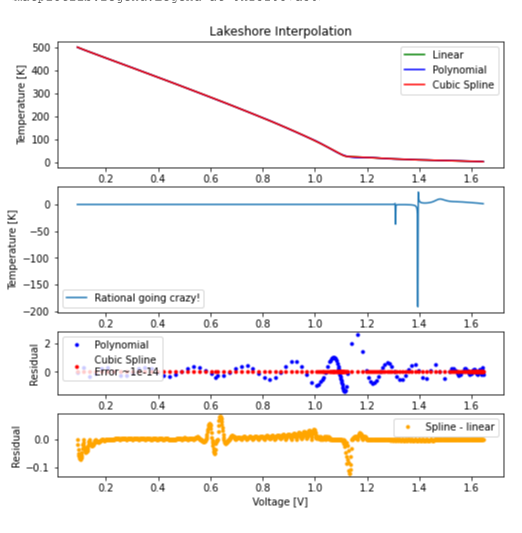
I wanted to test different interpolations (i.e. linear, rational, polynomial and cubic spline) to the data. The code block below is the implemented functions for each interpolation type. Without surprise, the cubic spline fit provided the best fit (judged visually). I thought that the rational interpolation would provide a good fit, but the length of the data is very long which requires very high numerator and denominator order and returns a crazy interpolation. I wondered if it would be best to try to fit for every some interval and then add the solutions but since the cubic spline provides a very good interpolation, I decided to use it to interpolate the temperatures.

In [9]:
```
#Following are the fits we learned in class


#Linear Fit
def linear_eval(x,y,point):
    # This interpolation "draws" a linear line between each point. Then, to interpolate at a point,
    #the two closest
    # points (a and b) are found and the returned value is interpolated from the "drawn" line between (a and b)

    # The points interpolated that are exactly = some xi value (from the data) will always have
    #a value exactly = yi
    myfun=interp.interp1d(x,y,'linear')
    return myfun(point)

#Polynomial Fit
def poly_eval(x_data,y_data,n,point):

    pp=np.polyfit(x_data,y_data,n) #numpy's polynomial fitter
    #Numpy description:
    #Fit a polynomial of degree n to points (x_data, y_data). Returns the coefficients p that minimises
    #the squared error in the order n, n-1, … 0.
    return np.polyval(pp,point) #uses the coefficients pp and evaluates the polynomial for each point

#Rational Fit
def rat_return(p,q,x):

    #These loops build the rational polynomials
    top=0
    #loop for numerator
    for i in range(len(p)):
        top=top+p[i]*x**i #multiply with coefficients, sum the terms
    bot=1
    #loop for denominator
    for i in range(len(q)):
        bot=bot+q[i]*x**(i+1) #multiply with coefficients, sum the terms
    return top/bot #R(x) = polynomial of numerator / polynomial of denominator

def rat_eval(x,y,n,m,point):

    #This function computes the coefficients
    assert(len(x)==n+m-1) #checking that the sum of the orders-1 corresponds to the length of the data to
    #interpolate
    assert(len(y)==len(x))

    mat=np.zeros([n+m-1,n+m-1]) #n+m-1 by n+m-1 matrix which consists of the elements from y(X) - y(X)qq(X)
    for i in range(n):
        mat[:,i]=x**i
    for i in range(1,m):
        mat[:,i-1+n]=-y*x**i

    pars=np.dot(np.linalg.pinv(mat),y) #inverse the matrix and multiply the entries with y's,
    #the results are p and q
    p=pars[:n]
    q=pars[n:]


    return rat_return(p,q,point)

def rat_eval2(x,y,n,m,point):

    #same function as above, but uses np.linalg.inv instead of np.linalg.pinv (see problem 4 for
    #comparison between the two)
    assert(len(x)==n+m-1)
    assert(len(y)==len(x))
    mat=np.zeros([n+m-1,n+m-1])
    for i in range(n):
        mat[:,i]=x**i
    for i in range(1,m):
        mat[:,i-1+n]=-y*x**i

    pars=np.dot(np.linalg.inv(mat),y)
    p=pars[:n]
    q=pars[n:]

    return rat_return(p,q,point)

#Cubic Spline fit
def cubspline_eval(x,y,point):

    spln=interp.splrep(x,y) #Approximates a smooth cubic spline; returns the vector of knots,
    #the B-spline coefficients, and the degree of the spline (3 since it's cubic)
    return interp.splev(point,spln) #Evaluates the spline at the points using the vector of knots
    #and the B-spline coefficients
```

In [10]:
```
xx = np.linspace(voltages[0],voltages[-1],1001)
#Linear Interp
interp_lin = linear_eval(voltages,temperatures,xx)

#Polynomial Interp
n=50; m=95
interp_pol = poly_eval(voltages,temperatures,n+m-1,xx)
y_pol = poly_eval(voltages,temperatures,n+m-1,voltages)

#rational Interp
n=50; m=95
interp_rat = rat_eval(voltages,temperatures,n,m,xx)

#Cubic Spline Interp
interp_cub = cubspline_eval(voltages,temperatures,xx)
y_cub = cubspline_eval(voltages,temperatures,voltages)
```

In [18]:
```
fig,axs = plt.subplots(4,figsize=[8,8],gridspec_kw={'height_ratios': [2,2,1,1]})
axs[0].plot(xx,interp_lin,color='g',label='Linear')
axs[0].plot(xx,interp_pol,color='b',label='Polynomial')
axs[0].plot(xx,interp_cub,color='r',label='Cubic Spline')
axs[0].legend()
axs[0].set_ylabel('Temperature [K]')
axs[0].set_title('Lakeshore Interpolation')
axs[1].plot(xx,interp_rat,label='Rational going crazy!')
axs[1].set_ylabel('Temperature [K]')
axs[1].legend()
axs[2].plot(voltages,temperatures-y_pol,'.',color='b',label='Polynomial')
axs[2].plot(voltages,temperatures-y_cub,'.',color='r',label='Cubic Spline\nError ~{}'\
            .format(10**np.round(np.log10(np.std(np.abs(temperatures-y_cub))))))
axs[2].set_ylabel('Residual')
axs[2].set_xlabel('Voltage [V]')
axs[2].legend()
axs[3].plot(xx,interp_cub-interp_lin,'.',color='orange',label='Spline - linear')
axs[3].set_xlabel('Voltage [V]')
axs[3].set_ylabel('Residual')
axs[3].legend()
```

Out[18]:
```
<matplotlib.legend.Legend at 0x13f057a30>
```

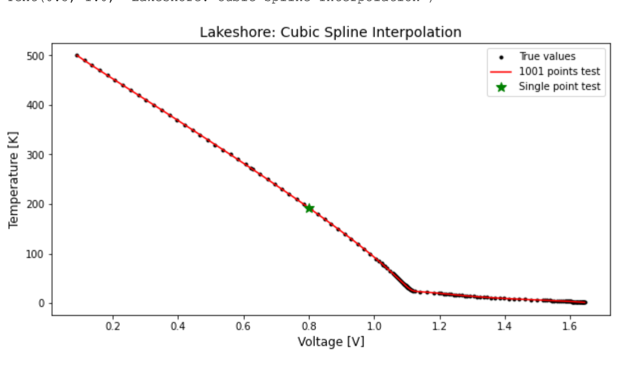


The linear residuals were not plotted on the the residual plot since they'd be computed as temperatures(voltages) - linearfit(voltages) which is always 0 with this type of interpolation. From the residuals, the cubic spline is much better than the polynomial (as expected). Also, from the last plot which shows the difference between the spline and the linear fit, we observe that the linear is also pretty good as the difference is small suggesting that the linear fit could have been used in the interpolation as well (with estimated errors ~1e-13). With that being said, the cubic spline is chosen to interpolate the data as it provides a smooth fit.

In [12]:
```
def lakeshore(V,data):

    temps = np.flip(data[:,0])
    volts = np.flip(data[:,1])

    #First, use the cubic spline fit
    interp = cubspline_eval(volts,temps,V)

    #Now evaluate error estimate
    temps_pred = cubspline_eval(volts,temps,volts)
    errors = np.abs(temps-temps_pred)
    #taking the standard deviation of the errors to estimate the error (inspired technique from John's)
    stds = 10**np.round(np.log10((np.std(errors)))) #rounding the errors to the nearest power of
    #10 as they are very small

    return [interp,stds]
```

In [13]:
```
#Testing the interpolation
x=0.8
xx=np.linspace(voltages[0],voltages[-1],1001)
test1=lakeshore(xx,data)
test2=lakeshore(x,data)
fig=plt.figure(figsize=(10,5))
plt.plot(voltages,temperatures,'o',markersize=3,color='k',label='True values')
plt.plot(xx,test1[0],color='r',label='1001 points test')
plt.plot(x,test2[0],'*',markersize=10,color='g',label='Single point test')
plt.legend()
plt.xlabel('Voltage [V]',fontsize=12)
plt.ylabel('Temperature [K]',fontsize=12)
plt.title('Lakeshore: Cubic Spline Interpolation',fontsize=14)
```

Out[13]:
```
Text(0.5, 1.0, 'Lakeshore: Cubic Spline Interpolation')
```

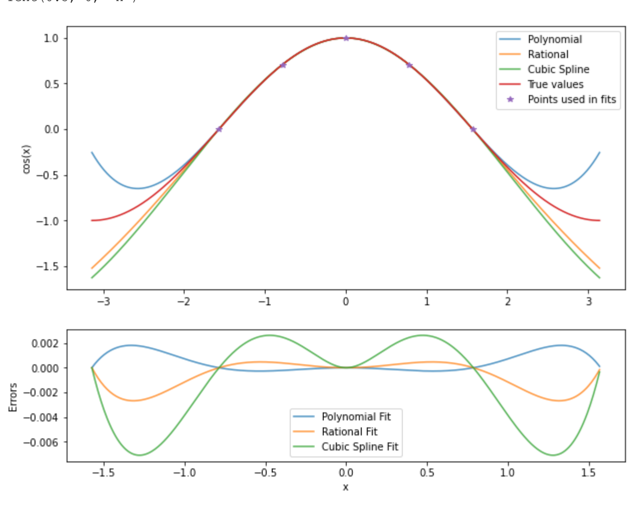


PROBLEM 4

The polynomial, cubic spline and rational interpolations are already defined in a block above so I used them here too.

In [14]:
```
n=3;m=3 #Rational order of n=3 and m=3, I use the order 5 for the polynomial to remain consistent.
#5 cos points are used.

x_points = np.linspace(-np.pi/2,np.pi/2,n+m-1) #5 x points
y_points = np.cos(x_points) #cos(x) of the 5 points

x_fits = np.linspace(-np.pi,np.pi,1001) # x points used to make the fits, interval is made longer
#to observe the tail
ind1 = np.where(x_fits==-np.pi/2)[0][0] # indices used when we determine the accuracy (s.t. the interval
#is from -pi/2 and pi/2)
ind2 = np.where(x_fits==np.pi/2)[0][0]
y_true = np.cos(x_fits[ind1:ind2]) #true values of the large array, used to determine the accuracy


#Polynomial Fit
y_poly = poly_eval(x_points,y_points,n+m-1,x_fits)

#Rational Fit
y_rat = rat_eval(x_points,y_points,n,m,x_fits)

#Cubic Spline Fit
y_cubspln = cubspline_eval(x_points,y_points,x_fits)

fig,axs=plt.subplots(2,figsize=(10,8),gridspec_kw={'height_ratios': [2,1]})
axs[0].plot(x_fits,y_poly,label='Polynomial',alpha=0.8)
axs[0].plot(x_fits,y_rat,label='Rational',alpha=0.8)
axs[0].plot(x_fits,y_cubspln,label='Cubic Spline',alpha=0.8)
axs[0].plot(x_fits,np.cos(x_fits),label='True values')
axs[0].plot(x_points,y_points,'*',label='Points used in fits')
axs[0].legend()
axs[0].set_ylabel('cos(x)')

axs[1].plot(x_fits[ind1:ind2],y_true-y_poly[ind1:ind2],label='Polynomial Fit',alpha=0.8)
axs[1].plot(x_fits[ind1:ind2],y_true-y_rat[ind1:ind2],label='Rational Fit',alpha=0.8)
axs[1].plot(x_fits[ind1:ind2],y_true-y_cubspln[ind1:ind2],label='Cubic Spline Fit',alpha=0.8)
axs[1].legend()
axs[1].set_ylabel('Errors')
axs[1].set_xlabel('x')
```

Out[14]:
```
Text(0.5, 0, 'x')
```

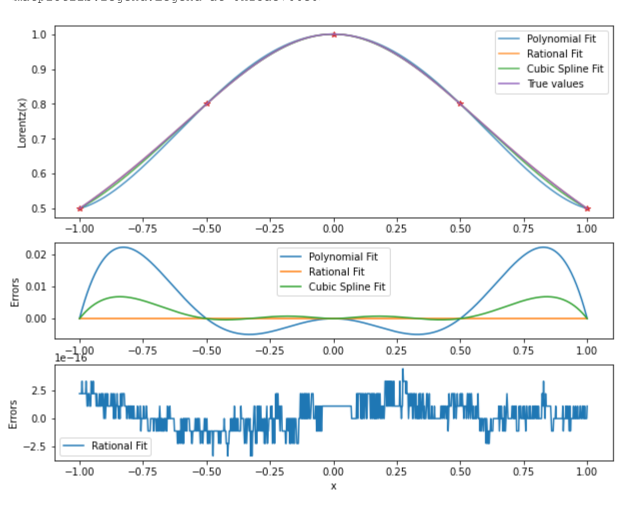


We observe that the polynomial and rational provide the best interpolations as cosine is essentially a taylor series. The cubic spline is a poorer fit

Trying with the Lorentzian now

In [15]:
```
def lorentz(x): #Lorentz function
    return 1/(1+x**2)

x_points = np.linspace(-1,1,n+m-1) #5 x points in the interval [-1,1]
y_points = np.array([lorentz(x) for x in x_points]) #Lorentz points

x_fits = np.linspace(-1,1,1001) # array used to make the interpolations
y_true = np.array([lorentz(x) for x in x_fits]) #true values

#Polynomial Fit
y_poly = poly_eval(x_points,y_points,n+m-1,x_fits)

#Rational Fit
y_rat = rat_eval(x_points,y_points,n,m,x_fits)

#Cubic Spline Fit
y_cubspln = cubspline_eval(x_points,y_points,x_fits)


fig,axs = plt.subplots(3,figsize=(10,8),gridspec_kw={'height_ratios': [2,1,1]})
axs[0].plot(x_fits,y_poly,label='Polynomial Fit',alpha=0.8)
axs[0].plot(x_fits,y_rat,label='Rational Fit',alpha=0.8)
axs[0].plot(x_fits,y_cubspln,label='Cubic Spline Fit',alpha=0.8)
axs[0].plot(x_points,y_points,'*')
axs[0].plot(x_fits,[lorentz(x) for x in x_fits],label='True values')
axs[0].legend()
axs[0].set_ylabel('Lorentz(x)')

axs[1].plot(x_fits,y_true-y_poly,'-',label='Polynomial Fit')
axs[1].plot(x_fits,y_true-y_rat,'-',label='Rational Fit')
axs[1].plot(x_fits,y_true-y_cubspln,'-',label='Cubic Spline Fit')
axs[1].legend()
axs[1].set_ylabel('Errors')

axs[2].plot(x_fits,y_true-y_rat,'-',label='Rational Fit')
axs[2].set_ylabel('Errors')
axs[2].set_xlabel('x')
axs[2].legend()
```

Out[15]:
```
<matplotlib.legend.Legend at 0x13ac766e0>
```

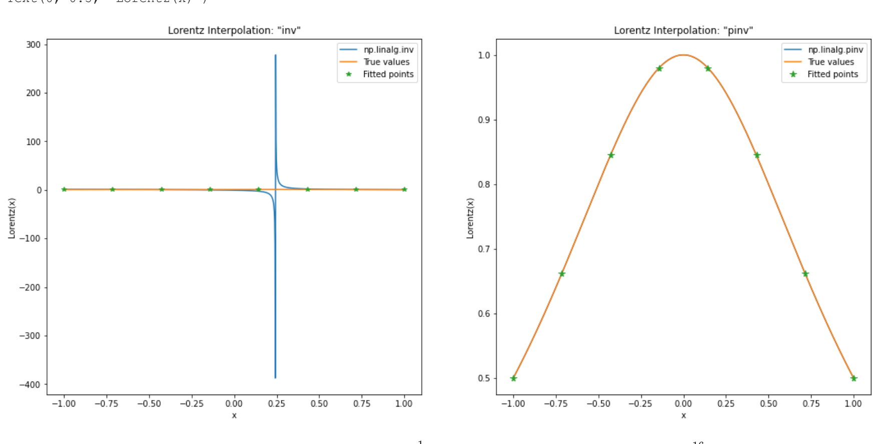


As expected, the rational interpolation yields the best fit since a lorentzian is a rational in itself. The accuracy of the rational interpolation is indeed as good as the machine accuracy (so the error = maching rounding error)!

Increasing the orders now:

In [16]:
```
#The code is similar as above

n=4;m=5 #increased order
x_points = np.linspace(-1,1,n+m-1)
y_points = np.array([lorentz(x) for x in x_points])

x_fits = np.linspace(-1,1,1001)
y_true = np.array([lorentz(x) for x in x_fits])


#Rational fit with np.linalg.pinv
y_rat1 = rat_eval(x_points,y_points,n,m,x_fits)

#Rational fit with np.linalg.inv
y_rat2 = rat_eval2(x_points,y_points,n,m,x_fits)

fig,axs = plt.subplots(1,2,figsize=(18,8))

axs[0].plot(x_fits,y_rat2,label='np.linalg.inv')
axs[0].plot(x_fits,y_true,label='True values')
axs[0].plot(x_points,y_points,'*',label='Fitted points')
axs[0].legend()
axs[0].set_title('Lorentz Interpolation: "inv"')
axs[0].set_xlabel('x')
axs[0].set_ylabel('Lorentz(x)')

axs[1].plot(x_fits,y_rat1,label='np.linalg.pinv')
axs[1].plot(x_fits,y_true,'-',label='True values')
axs[1].plot(x_points,y_points,'*',markersize=8,zorder=3,label='Fitted points')
axs[1].legend()
axs[1].set_title('Lorentz Interpolation: "pinv"')
axs[1].set_xlabel('x')
axs[1].set_ylabel('Lorentz(x)')
```

Out[16]:
```
Text(0, 0.5, 'Lorentz(x)')
```



We would expect the interpolation to replicate the lorentzian ($\frac{1}{1+x^2}$) "exactly" (with machine error (i.e. 10^{-16}) since the function is also a rational function. However, we observe that the rational interpolation with np.linalg.inv goes crazy when the order of the denominator (m) is greater than 3 and the order of the numerator (n) is greater than 1. This is due to exceeding terms in the rational $R(x)$ (i.e. polynomial of degree higher than 2 (m>3) in the denominator and higher than 0 (n>1) in the numerator). In this case, $R(x)$ can be written as $R(x) = \frac{p(x)(1+ax)(1+bx)...(1+cx)}{(1+qq(x))(1+ax)(1+bx)...(1+cx)}$, where each factor $(1+\lambda x)$ get cancel for $R(x)$ to eventually simplify to a lorentzian (where the coefficients of the extra terms in the numerator or in the denominator (depending on n and m) are set to 0 by the interpolation). Then, when the inverse of the matrix is taken to compute the coefficients of p(x) and q(x), the cancelling factors will yield values resulting in "interpolation craziness", which is what happens when np.linalg.inv is used to invert the matrix. On the other hand, np.linalg.pinv returns the pseudo inverse by using its singular-value decomposition. This way, the rational will give an accurate interpolation of the Lorentzian, by accounting for the unexpected results that occur due to the cancelling factors.

In a few words, the condition for the rational interpolation with np.linalg.pinv to fail is to have at least one cancelling factor (which occurs if m>3 and n>1).