Computational physics -- problem set 1

My work (code and math) for problem set 1 of PHYS512.

In [4]:
```
# import all the fun stuff

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("ticks")

from scipy.interpolate import CubicSpline

%matplotlib inline
```

Problem 1

1.a)

The correct combination of functions will have to be "symmetric", i.e. come in pairs with $f(x+\delta) - f(x-\delta)$. This is required in order to cancel the δ^2 term, which leads to a $\mathcal{O}(\delta)$ error. I claim that the correct combination is

$$\boxed{f^{'}(x) = \frac{8f(x+\delta) - f(x+2\delta) + f(x-2\delta) - 8f(x-\delta)}{12\delta}}. \tag{1}$$

Taylor expanding each term, one indeed finds

$$f^{'}(x) = f^{'}(x) - \frac{\delta^4}{30} f^5(x) + \mathcal{O}(\delta^6) \tag{2}$$

where f^5 is the fifth derivative of f. Thus, not only have we taken the correct combination to cancel out the third-order terms, but we also cancel all even order terms for free due to the centred nature of the derivative (i.e. terms like. $f(x+\delta) - f(x-\delta)$ will not have even terms in their Taylor series).

1.b)

To find the optimal δ as a function of machine precision, we proceed in the exact same way as done by Jon in class. That is, we multiply each term by a factor $1+\epsilon g$ where ϵ is the machine precision and g is a random number o forder unity provided by the gods of randomness. In doing so, we arrive at

$$f^{'}(x) = f^{'}(x) + \frac{f(x)\epsilon g}{\delta} + \frac{f^5(x)\delta^4}{30} \equiv f^{'}(x) + \text{Error}(\delta), \tag{3}$$

where we have ommitted all higher order terms. To find the optimal δ, simply minimize the error term. This yields

$$\boxed{\delta \sim (\frac{f}{f^5}\epsilon)^{1/5}}. \tag{4}$$

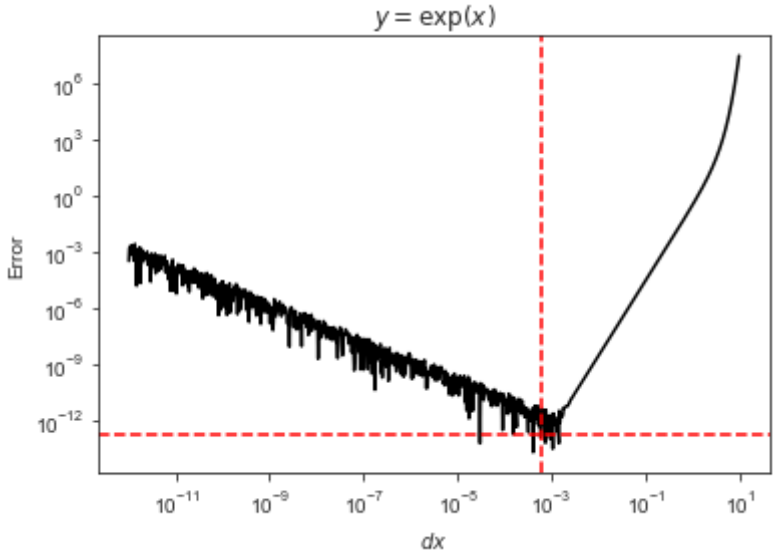
So, for e^x the optimal δ is $\sim \epsilon^{1/5}$, while for $e^{0.01x}$ it is $\sim \epsilon^{1/5}/0.01$ (the extra $1/0.01$ comes from the derivative term).

In [89]:
```
def ndiff2(fun,x,dx):
    """
    Numerical derivative using both +/- dx and +/- 2*dx
    """
    # compute the function at the points of interest
    yplus = fun(x + dx)
    yminus = fun(x - dx)
    yplus2 = fun(x + 2*dx)
    yminus2 = fun(x - 2*dx)

    # compute the numerical derivative
    fprime = (8 * yplus - yplus2 + yminus2 - 8 * yminus) / (12 * dx)

    return fprime
```

In [120…
```
# set up logspace
dxlog = np.linspace(-12,1,num=1000)
dxs = 10**(dxlog)

# make it interesting
x0 = 2.1

# functions we'll be trying out
fun_exp = np.exp
fun_exp_mod = lambda x: np.exp(0.01 * x)

# run it!
fprime_exp_dx = ndiff2(fun_exp,x0,dxs)
fprime_exp_mod_dx = ndiff2(fun_exp_mod,x0,dxs)

# compute the errors as a test of validity
error_exp_dx = np.abs(fprime_exp_dx - np.exp(x0))
error_exp_mod_dx = np.abs(fprime_exp_mod_dx - 0.01*np.exp(0.01*x0))
```

For e^x:

In [128…
```
plt.clf()

# plot fprime error as a function of dx
plt.loglog(dxs, error_exp_dx, c='k')

dx_optimal = 10 ** (-16/5)
estimated_error = dx_optimal ** (4)

plt.axvline(dx_optimal, c='r', ls='--', label="Optimal dx")
plt.axhline(estimated_error, c='r', ls='--', label="Estimated error")

plt.title("$y = \exp{(x)}$")
plt.xlabel("$dx$")
plt.ylabel("Error")

plt.show()
```

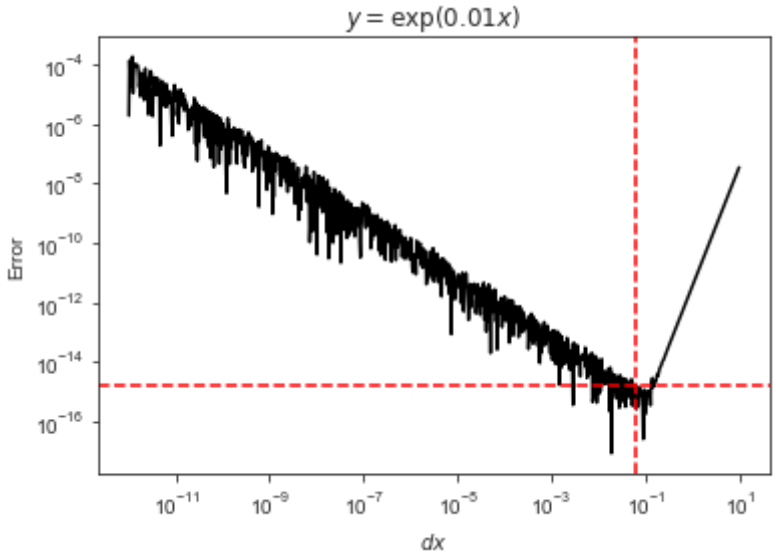


Here, the vertical (horiztonal) dashed line denotes the optimal dx (estimated error) from our calculations.

For $e^{0.01x}$:

In [127…

```
plt.clf()

# plot fprime error as a function of dx
plt.loglog(dxs, error_exp_mod_dx, c='k')

# we have to take into account the f/f^5 factor now
# of course, if we knew the derivative we wouldn't be using numerical tools
# but this is for the sake of comparison
dx_optimal_mod = 10 ** (-16/5) * (1/(0.01) ** 5) ** (1/5)
estimated_error_mod = dx_optimal_mod ** (4) * (0.01) ** 5

plt.axvline(dx_optimal_mod, c='r', ls='--', label="Optimal dx")
plt.axhline(estimated_error_mod, c='r', ls='--', label="Estimated error")

plt.title("$y = \exp{(0.01 x)}$")
plt.xlabel("$dx$")
plt.ylabel("Error")

plt.show()
```



We see that our results are (at least roughly) correct.

Problem 2

In order to estimate the optimal dx, we need to estimate the third derivative of a given function. Using the same Taylor series tricks, we can see that the combination of functions

$$f''' = \frac{f(x+2dx) - 2f(x+dx) + 2f(x-dx) - f(x-2dx)}{2dx^3} \tag{5}$$

cancels the first derivative (and all even derivatives by symmetry). The leading error is $dx^5/dx^3 = dx^2$, which is not great but decent for a rough approximation.

In [104…

```
def ndiff(fun,x,full=False):
    """
    Compute the numerical derivative of function `fun` at points `[x]`.
    If `full`, returns the derivative, dx, and an estimate on the error of deriv
    To deal with arrays, `fun` must be a numpy function and x must be a numpy nd
    """
    # set the initial guess for dx to be ~ 10^{-3}
    # if we set it to 10^{-16/3} (as we may be tempted to do),
    # we will be dividing by machine precision to estimate the third derivative,
    # which is not good!
    dx = 10**(-3)

    # compute the function at the points of interest
    yplus = fun(x + dx)
    yminus = fun(x - dx)

    # we need these more points to estimate the optimal dx
    yplus2 = fun(x + 2 * dx)
    yminus2 = fun(x - 2 * dx)

    # what is the optimal error? precisely, it is ~(f/f''' * epsilon)^{1/3}
    # since we do not know f''', we must estimate it using a fixed dx
    f_0 = fun(x)
    f_3 = (yplus2 - 2 * yplus + 2 * yminus - yminus2) / (2 * dx**3)

    # optimal dx is therefore
```

```
    dx_opt = np.abs(f_0 / f_3) ** (1/3) * (10**(-16/3)) # sign is irrelevant

    # compute the numerical derivative with the new dx_optimal
    fprime = (fun(x + dx_opt) - fun(x - dx_opt)) / (2 * dx_opt)

    error = f_3 * dx_opt**2 # leading term in the Taylor series is ~ f'''*dx^2

    if full:
        return fprime, dx_opt, error

    return fprime
```

Let's run some quick tests for different functions. Namely, $y = e^x, \sin x, \arctan x$ with the analytic derivatives $y^{'} = e^x, \cos x, \frac{1}{1+x^2}$, respectively.

In [105…
```
# functions to test
fun1 = np.exp
fun2 = np.sin
fun3 = np.arctan

# their true derivatives
true1 = np.exp
true2 = np.cos
true3 = lambda x: np.divide(1,np.power(x,2)+1)

# the array we'll be using to verify that it does indeed work with x as an array
xs = np.array([-1,0.02,np.e])

# run it!
returns1, returns2, returns3 = [ndiff(fun1,xs,full=True), ndiff(fun2,xs,full=Tru
fprime1, dx1, error1 = returns1
fprime2, dx2, error2 = returns2
fprime3, dx3, error3 = returns3
```

To check if our code worked, let's compare it to the true functions and see if the error does in fact match (to an order of magnitude) the error we compute.

In [106…
```
delta1 = fprime1 - true1(xs)
delta2 = fprime2 - true2(xs)
delta3 = fprime3 - true3(xs)

print("The derivative error of np.exp is {}. Our estimated error is {} for all p
print("The derivative error of np.sin is {}. Our estimated error is {} for all p
print("The derivative error of np.arctan is {}. Our estimated error is {} for al
```

```
The derivative error of np.exp is [-4.06780165e-12  2.30526709e-12  5.77282222e-
10]. Our estimated error is [7.92572335e-12 2.19795750e-11 3.26488718e-10] for a
ll points.
The derivative error of np.sin is [ 7.58937357e-12 -9.51128065e-13  2.92178504e-
11]. Our estimated error is [-1.56400568e-11 -1.58722454e-12  1.15441804e-11] fo
r all points.
The derivative error of np.arctan is [ 6.25222096e-12 -1.31361588e-12  9.3943741
7e-12]. Our estimated error is [ 1.45562574e-11 -1.99822073e-12  1.02097890e-11]
for all points.
```

As we can see, the estimated error is in good agreement with the true error. Moreover, the true error is quite small ($\sim 10^{-11}$) compared to the points of interest ($\sim 10^{1}$) and differential element dx (10^{-5}).

Problem 3

In [110…

```
# we'll use what's already here -- the built in CubicSpline function from scipy

def lakeshore(V, data):
    """
    Returns the interpolated temperature for a given voltage and fixed conversio
    The voltage V must be between the min and max of given voltage data -- i.e.,
    """
    # set up the arrays from the given data
    temperature = data[:,0]
    voltage = data[:,1]
    dvdt = data[:,2]

    # make sure that the voltages are in increasing order to use in CubicSpline
    idxs = voltage.argsort()
    voltage_fixed = voltage[idxs]
    temperature_fixed = temperature[idxs]

    # use CubicSpline to get the temperature
    temperature_smooth = CubicSpline(voltage_fixed,temperature_fixed, extrapolat

    # compute the temperature at the given points
    interpolated_temperature = temperature_smooth(V)

    # our rough error estimate is the difference between the
    # interpolated temperature and the closest voltage's temperature

    # first, find index of closest point to true data
    # to do so in a way which accomodates arrays,
    # we must change the size to find each given Voltage's
    # closest point to the given voltage

    # V_full's len(voltage_fixed) rows are V
    V_full = np.broadcast_to(V, (np.size(voltage_fixed),np.size(V)))

    # voltage_fixed_full's len(V) columns are voltage fixed
    voltage_fixed_full = np.broadcast_to(voltage_fixed, (np.size(V),np.size(volt

    # find the minimum index along the 0 axis, to be left with len(V)
    closest_idxs = np.argmin(np.abs(V_full - voltage_fixed_full), axis=0)
    # the error is the absolute difference between the true T value and the extr
    error = np.abs(interpolated_temperature - temperature_fixed[closest_idxs])

    # return the temperatures and the errors
    return interpolated_temperature, error
```

Let's test this code when V is a number:

In [111…

```
V_num1 = 1.3 # picked out of thin air

# import the data
lakeshore_data = np.loadtxt("lakeshore.txt")
temperature = data[:,0]
voltage = data[:,1]

T_num1, error_num1 = lakeshore(V=V_num1, data=data)

print("Interpolated T = {}. Error = {}".format(T_num1, error_num1))
```

```
Interpolated T = 13.70254427846655. Error = [0.20254428]
```

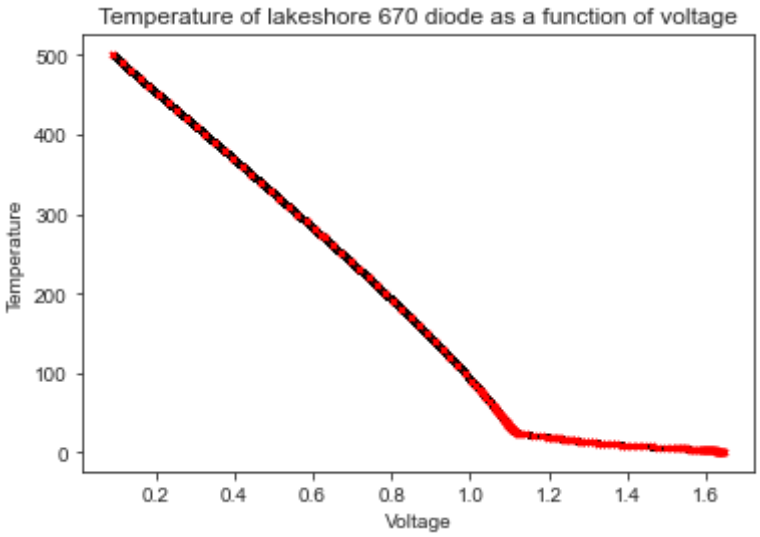
If the voltage we want is already in the given data set, the error will be zero (this is part of why we chose our particular rough error scheme to be the difference between the closest true voltage and the interpolated one):

In [112…
```
V_num2 = 0.949 # picked out of thin air

T_num2, error_num2 = lakeshore(V=V_num2, data=data)

print("Interpolated T = {}. Error = {}".format(T_num2, error_num2))
```

```
Interpolated T = 120.0. Error = [0.]
```

To get a better sense of how our interpolation works, we should plot and visualize it vis-a-vis the data points. This also requires us to treat V as an array, so we can show its validity in this regime as well.

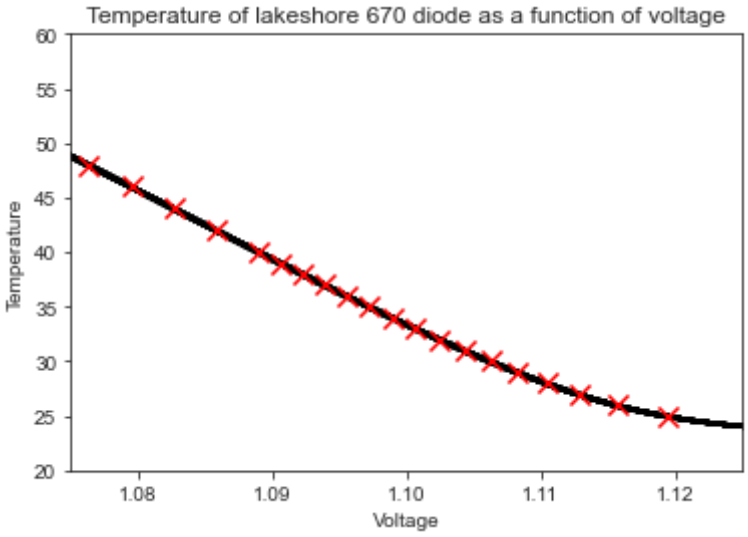
In [129…
```
num_smooth = 100000

V_smooth = np.linspace(np.min(voltage),np.max(voltage),num=num_smooth)

T_smooth, error_smooth = lakeshore(V_smooth, data)

plt.clf()

plt.scatter(V_smooth, T_smooth, marker='.', c='k', s=10)
plt.scatter(voltage, temperature, marker='x', c='r', s=10)

plt.title("Temperature of lakeshore 670 diode as a function of voltage")
plt.xlabel("Voltage")
plt.ylabel("Temperature")

plt.show()
```



Let's take a closer look in a smaller region:

In [130…
```
plt.clf()

plt.scatter(V_smooth, T_smooth, marker='.', c='k', s=10)
plt.scatter(voltage, temperature, marker='x', c='r', s=100)
```

```
plt.xlim(1.075,1.125)
plt.ylim(20,60)

plt.title("Temperature of lakeshore 670 diode as a function of voltage")
plt.xlabel("Voltage")
plt.ylabel("Temperature")

plt.show()
```



The interpolation works quite well, and is well behaved. To quantify this, we can compute the standard deviation of all errors, normalized to the number of points:

In [109…
```
print(np.std(error_smooth) / num_smooth)
```

```
1.58140891624566e-05
```

which is indeed quite small. So, we've developed a routine which can take an array of V s and interpolate (using a cubic spline algorithm) to find the value of temperature at any voltage.

Problem 4

For polynomial interpolation, we define the function

$$y_p(x) = \sum_{i=0}^{N} y_i b_i(x) \tag{6}$$

where $b_i(x)$ is

$$b_i(x) = \frac{(x - x_0)\dots(x - x_{i-1})(x - x_{i+1})\dots(x - x_N)}{(x_i - x_0)\dots(x_i - x_{i-1})(x_i - x_{i+1})\dots(x_i - x_N)} \tag{7}$$

a polynomial function equal to 1 a x_i and 0 at all other $x_{j\neq i}$ sampled points. Let's code it up!

In [150…
```
# slightly borrowing from Jon's code

# these are the points which we'll sample
order = 10 # modest number of points

x = np.linspace(-np.pi/2,np.pi/2,num=order)
```

```
y = np.cos(x)

# this is the smooth x space we'll use
smooth = 1000
xs = np.linspace(-np.pi/2,np.pi/2,smooth+1)
cos_smooth = np.cos(xs)
```

In [114…

```
# first, the polynomial fit

def b_i(xs,i,x,y,order):
    """
    xs is the smooth linspace, i is the index of the missing point, x and y are
    """
    # define x_i
    x_i = x[i]

    # make an array of sampled x without x_i
    x_no_i = np.hstack([x[:i],x[i+1:]])

    # loop over the polynomial factors (x - x_j)
    b = 1. # set the initial function value
    for j in range(len(x)-1):
        b = b * (xs - x_no_i[j]) / (x_i - x_no_i[j]) # make sure to normalize it

    return b

# now we need to sum over y_i * b_i
# define a polyfit function

def polyfit(xs,x,y):
    """
    Returns smooth interpolated y-values
    x and y are the sampled points
    """
    # set p = 0
    p = 0

    # sum over all points, leaving xs generic
    order = len(x)
    for i in range(order):
        p = p + y[i] * b_i(xs,i,x,y,order)

    return p
```

In [115…

```
# run it for our problem
poly = polyfit(xs,x,y)
```

Now, we'll do a cubic spline interpolation. We've already done this above, so it shouldn't be too hard.

In [116…

```
# use the built in function, and tweak it a bit for our purposes
def splinefit(xs,x,y):
    f = CubicSpline(x,y,extrapolate=False)
    return f(xs)

spline = splinefit(xs,x,y)
```

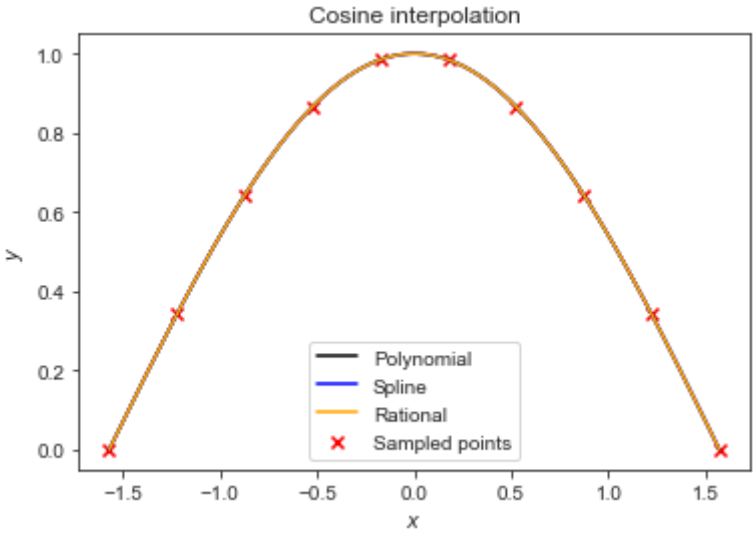
Finally, let's go for a rational function interpolation. This is similar to the polynomial, except it can handle functions which tail off at infinity, for example. Once again, we'll borrow some code from Jon.

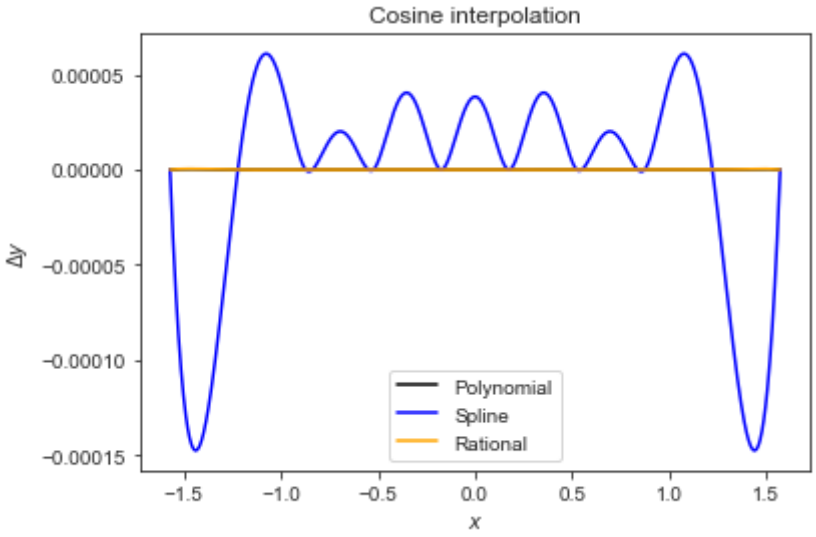
In [125…
```
def ratpolyfit(xs,x,y,m,n,order):
    """
    Returns a function for the rational-fit interpolated y-values given sample p
    m,n are the denominator and numerator orders, respectively
    """
    assert(m+n+1==order)

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

    # make q column vector
    qcols=[-x**k*y for k in range(1,m+1)]
    qmat=np.vstack(qcols)

    # the matrix to invert
    mat=np.hstack([pmat.T,qmat.T])
    coeffs=np.linalg.inv(mat)@y

    # numerator (we have to flip to entries because of how polyval takes it)
    num=np.polyval(np.flipud(coeffs[:n+1]),xs)

    # denominator (we need an extra x because of how we defined the problem)
    denom=1+xs*np.polyval(np.flipud(coeffs[n+1:]),xs)

    # take the ratio of the numerator / denominator
    rat=num/denom

    return rat
```

In [147…
```
rat = ratpolyfit(xs,x,y,m=5,n=4,order=order)
```

In [155…
```
# plot the results

plt.clf()

plt.plot(xs,poly,c='k',label='Polynomial')
plt.plot(xs,spline,c='blue',label='Spline')
plt.plot(xs,rat,c='orange',label='Rational')

plt.scatter(x,y,c='r',marker='x',label='Sampled points')

plt.title("Cosine interpolation")
plt.xlabel(r"$x$")
plt.ylabel(r"$ y$")

plt.legend()

plt.show()

# compute the error of each method

poly_error = cos_smooth - poly
spline_error = cos_smooth - spline
```

```
rat_error = cos_smooth - rat

# plot the error

plt.clf()

plt.plot(xs,poly_error,c='k',label='Polynomial')
plt.plot(xs,spline_error,c='blue',label='Spline')
plt.plot(xs,rat_error,c='orange',label='Rational')

plt.title("Cosine interpolation")
plt.xlabel(r"$x$")
plt.ylabel(r"$ \Delta y$")

plt.legend()

plt.show()
```





Now, we'll look at a Lorentzian $\frac{1}{1+x^2}$ between -1 and 1.

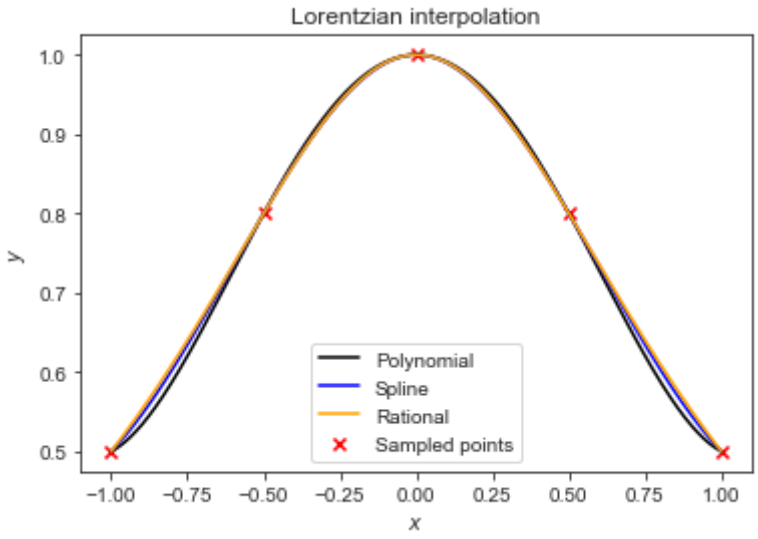
In [142…

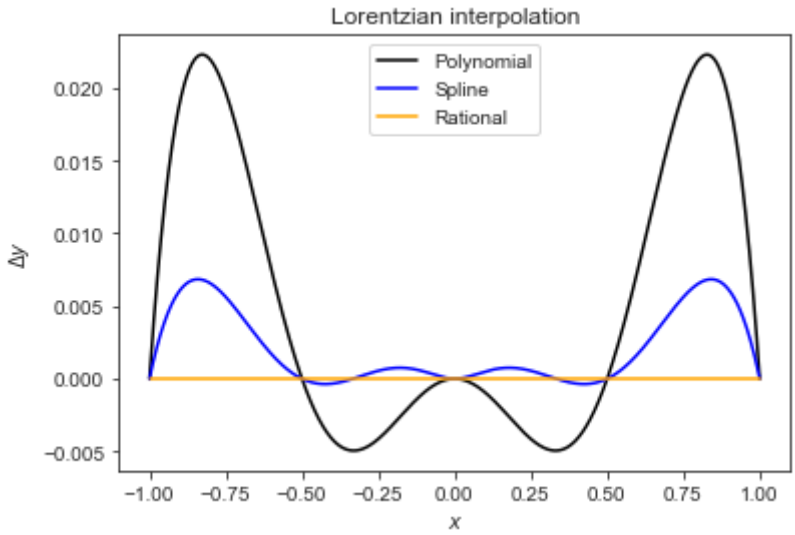
```
m = 2
n = 2
order_lor = m + n + 1 # modest number of points

def lorentzian(x):
```

```
    return 1 / (1 + x ** 2)

# make the sample points
x_lor = np.linspace(-1,1,order_lor)
y_lor = lorentzian(x_lor)

# make the smooth points
xs_lor = np.linspace(-1,1,smooth+1)
ys_lor = lorentzian(xs_lor)

# compute the fits
poly_lor = polyfit(xs_lor,x_lor,y_lor)
spline_lor = splinefit(xs_lor,x_lor,y_lor)
rat_lor = ratpolyfit(xs_lor,x_lor,y_lor,m,n,order=order_lor)
```

In [156…

```
# plot the fits

plt.clf()

plt.plot(xs_lor,poly_lor,c='k',label='Polynomial')
plt.plot(xs_lor,spline_lor,c='blue',label='Spline')
plt.plot(xs_lor,rat_lor,c='orange',label='Rational')

plt.scatter(x_lor,y_lor,c='r',marker='x',label='Sampled points')

plt.title("Lorentzian interpolation")
plt.xlabel(r"$x$")
plt.ylabel(r"$  y$")

plt.legend()

plt.show()

# compute the error of each method

poly_error_lor = ys_lor - poly_lor
spline_error_lor = ys_lor - spline_lor
rat_error_lor = ys_lor - rat_lor

# plot the error

plt.clf()

plt.plot(xs_lor,poly_error_lor,c='k',label='Polynomial')
plt.plot(xs_lor,spline_error_lor,c='blue',label='Spline')
plt.plot(xs_lor,rat_error_lor,c='orange',label='Rational')


plt.title("Lorentzian interpolation")
plt.xlabel(r"$x$")
plt.ylabel(r"$ \Delta y$")

plt.legend()

plt.show()
```





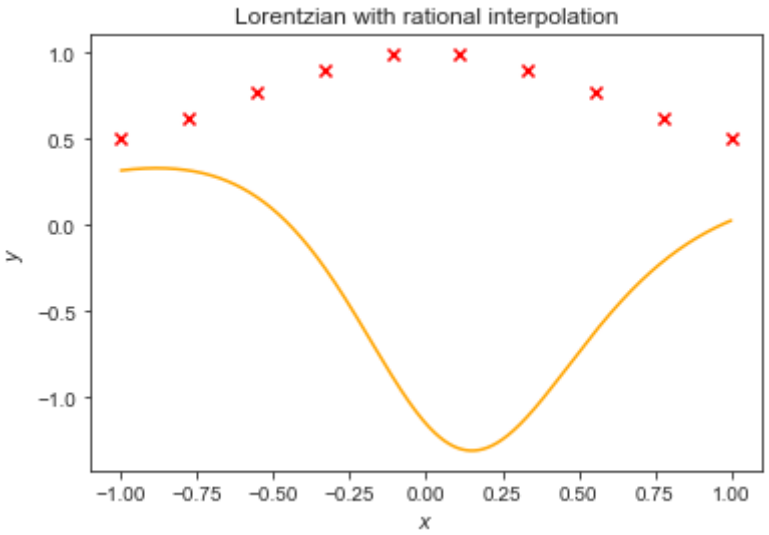
For the rational fit (orange line), the error should be zero since a Lorentzian is a rational function. This is in agreement for the orders $m = n = 2$.

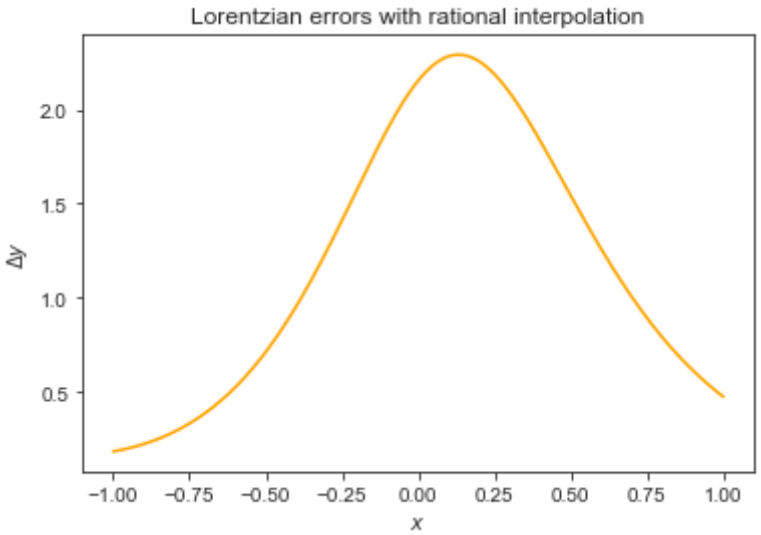
If we increase the order, something interesting happens. For $m = 5$, $n = 4$:

In [140…

```
m = 5
n = 4

order_lor = m + n + 1

x_lor = np.linspace(-1,1,order_lor)
y_lor = lorentzian(x_lor)

rat_lor = ratpolyfit(xs_lor,x_lor,y_lor,m=m,n=n,order=order_lor)

# plot the fits

plt.clf()

plt.plot(xs_lor,rat_lor,c='orange')

plt.scatter(x_lor,y_lor,c='r',marker='x')

plt.title("Lorentzian with rational interpolation")
```

```
plt.xlabel(r"$x$")
plt.ylabel(r"$ y$")

plt.show()

# compute the error of each method

rat_error_lor = ys_lor - rat_lor

# plot the error

plt.clf()

plt.plot(xs_lor,rat_error_lor,c='orange')


plt.title("Lorentzian errors with rational interpolation")
plt.xlabel(r"$x$")
plt.ylabel(r"$\Delta y$")

plt.show()
```



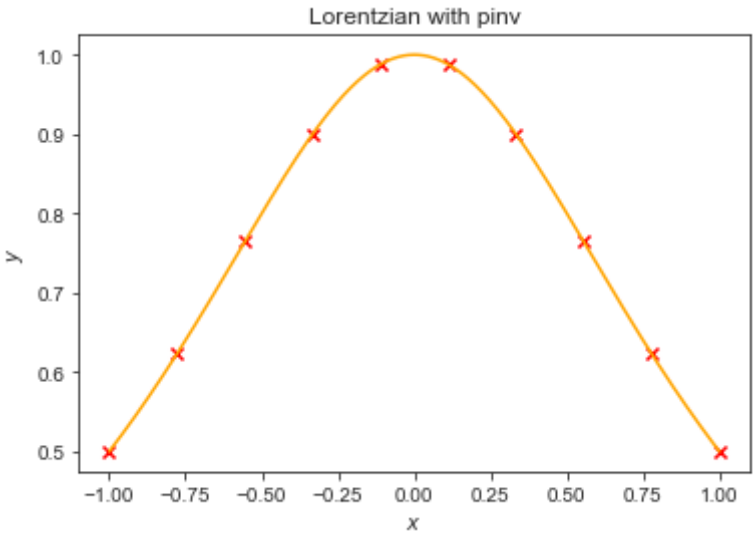


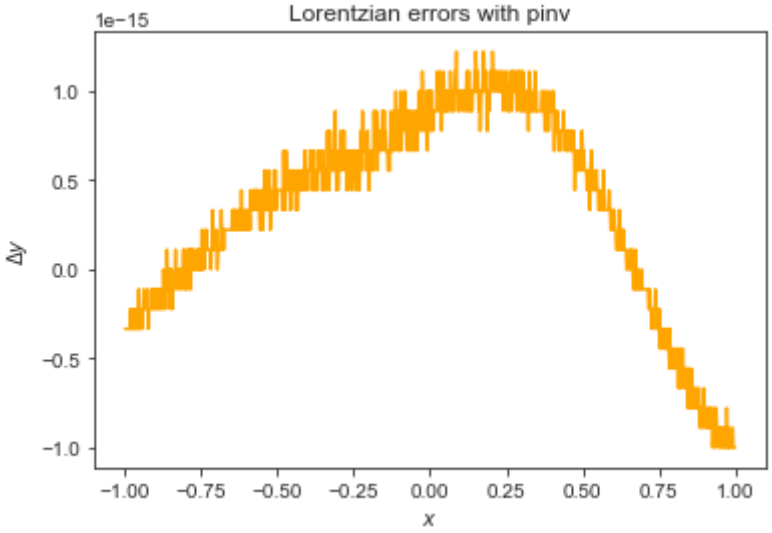
The answer is completely bogus! The interpolation does not agree with our intuition that there should be no error, clearly. If we change `inv` to `pinv`, the problem is fixed:

In [131…
```
def ratpolyfit_fix(xs,x,y,m,n,order):
    """
    Returns a function for the rational-fit interpolated y-values given sample p
    m,n are the denominator and numerator orders, respectively
    """
    assert(m+n+1==order)

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

    # make q column vector
    qcols=[-x**k*y for k in range(1,m+1)]
    qmat=np.vstack(qcols)

    # the matrix to invert
    mat=np.hstack([pmat.T,qmat.T])
    coeffs=np.linalg.pinv(mat)@y # <--- note the .pinv and not .inv

    # numerator (we have to flip to entries because of how polyval takes it)
    num=np.polyval(np.flipud(coeffs[:n+1]),xs)

    # denominator (we need an extra x because of how we defined the problem)
    denom=1+xs*np.polyval(np.flipud(coeffs[n+1:]),xs)

    # take the ratio of the numerator / denominator
    rat=num/denom

    return rat, coeffs
```

In [137…
```
m = 5
n = 4

order_lor = m + n + 1

x_lor = np.linspace(-1,1,order_lor)
y_lor = lorentzian(x_lor)

rat_lor, coeffs = ratpolyfit_fix(xs_lor,x_lor,y_lor,m=m,n=n,order=order_lor)

# plot the fits

plt.clf()

plt.plot(xs_lor,rat_lor,c='orange')

plt.scatter(x_lor,y_lor,c='r',marker='x')

plt.title("Lorentzian with pinv")
plt.xlabel(r"$x$")
plt.ylabel(r"$ y$")

plt.show()

# compute the error of each method

rat_error_lor = ys_lor - rat_lor

# plot the error
```

```
plt.clf()

plt.plot(xs_lor,rat_error_lor,c='orange')

plt.title("Lorentzian errors with pinv")
plt.xlabel(r"$x$")
plt.ylabel(r"$\Delta y$")

plt.show()
```





The error is now on the machine precision (10^{-16}) scale. What happened? What's the difference? It all comes down to how our algorithm deals with matrix inverses. In the first example, it took the inverse -- no frills. This can cause some problems when the eigenvalues of the original matrix are zero, since the inverse of zero is infinity. What `pinv` does is it sets those zeros to zero in the inverted matrix, thus "dealing" with the singularities. And, since the Lorentzian is \textit{exactly} a rational function, all but two coefficients (p_0 and qq_1) will be zero. Inverse of zero is large, and we've got a problem. What `pinv` does is it circumvents this by setting those inverse eigenvalues to zero, which is what they should be analytically (exactly how the conductivity and its inverse can both have zero eigenvalues because of their matrix structure).

To be more precise about this, recall our rational function

$$y = \frac{p(x)}{1 + qq(x)}. \tag{8}$$

Now say we want to multiply this by one via $(1 + ax)/(1 + ax)$ to see how our algorithm deals with singularities. While \textit{we} know that the terms cancel, the machine certainly doesn't, and moreover, the rational function's coefficients ($p(x) = p_0 + p_1 x + \ldots$) now change since $p(x) \to p(x) + axp(x) \equiv \tilde{p}(x)$ and similarly for $qq(x)$! Indeed, the coefficients are not all 1 and 0 anymore:

In [134…
```
print(coeffs)
```

```
[ 1.00000000e+00  1.99840144e-15 -3.33333333e-01  6.66133815e-16
 -3.55271368e-15  3.10862447e-15  6.66666667e-01 -1.33226763e-15
 -3.33333333e-01  2.66453526e-15]
```

We have 1 and 0 still, but we also have $2/3$ and $-1/3$. The coefficients have indeed changed due to the singular nature of the problem, which stems from trying to interpolate an \textit{exact} rational function with rational polynomial interpolation.

In [ ]: