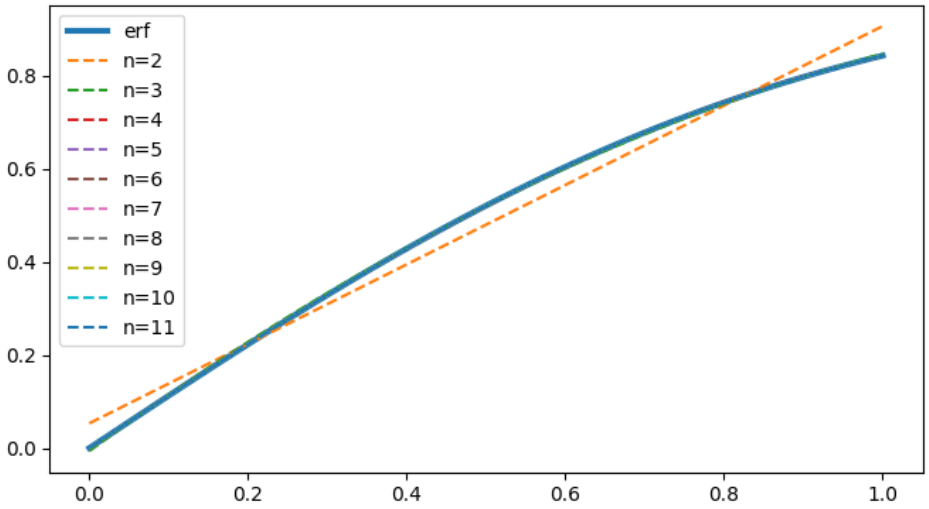
All code runs using python3. With numpy, scipy and matplotlib as prerequisites.

1 *Code is at one.py - at the end of the file there are invocations for all sub questions.*

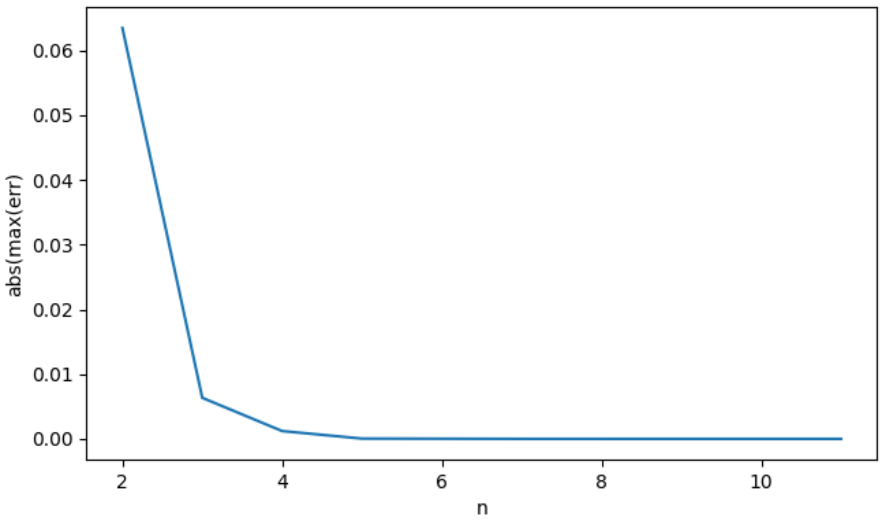
(a)

At one.py: `fit(pol)` plots the following result:
With `n` being the number of parameters and the degree being actually `n - 1`



With the output for error and “cost”:

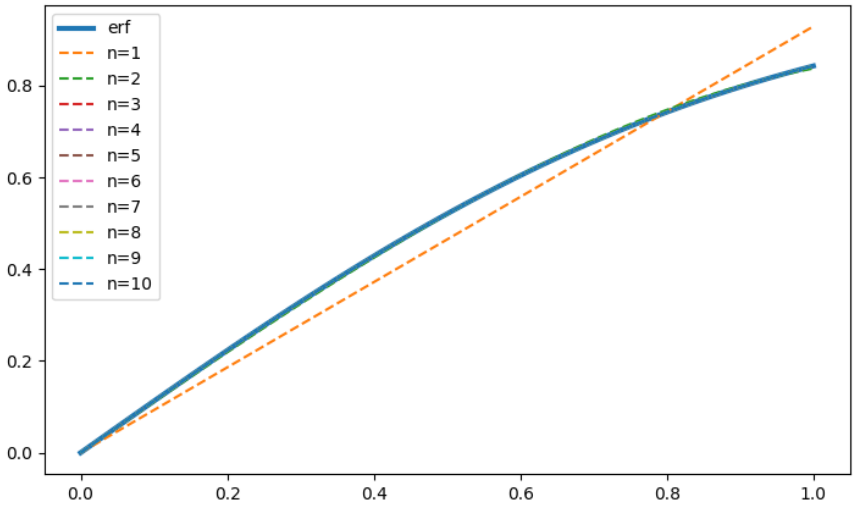
```
n=2, max_err=0.063454,
cost=0.0068134
n=3, max_err=0.0063455,
cost=6.9436e-05
n=4, max_err=0.00121,
cost=5.6272e-06
n=5, max_err=4.6595e-05,
cost=2.4794e-09
n=6, max_err=2.4099e-05,
cost=1.3829e-09
n=7, max_err=7.0039e-07,
cost=6.7197e-13
n=8, max_err=1.9388e-07,
cost=9.2346e-14
n=9, max_err=9.8422e-09,
cost=1.2046e-16
n=10, max_err=2.2361e-09,
cost=7.9526e-18
n=11, max_err=1.0166e-09,
cost=1.336e-18
```



Indeed the error from the target function of err grows smaller as the degree of the polynomial grows as well. Even for n=2 the result is pretty good as at this range the error func is almost linear.
For n=3 (the red) it gets a very nice curve and a small error.

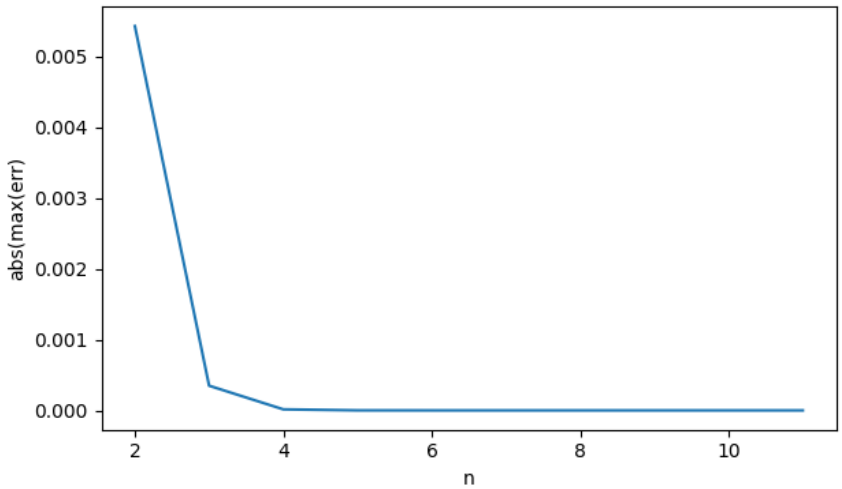
(b)

Passing a different function to `fit()`. Namely running `fit(pol_odd)` gives the following plot and errors (with changing the range(2, 12) at line 36 to range(1, 11)):
With actual polynomial degree `2n - 1`



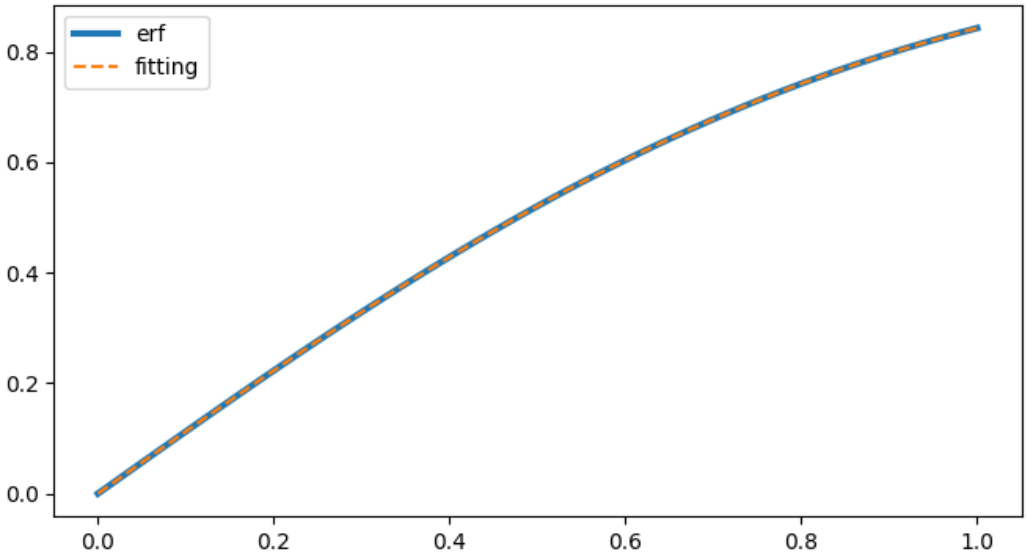

```
n=1, max_err=0.086223, cost=0.011249
n=2, max_err=0.0054263, cost=6.4264e-05
n=3, max_err=0.00034862, cost=2.1827e-07
n=4, max_err=1.473e-05, cost=4.3431e-10
n=5, max_err=5.4502e-07, cost=5.0096e-13
n=6, max_err=1.376e-08, cost=3.2206e-16
n=7, max_err=2.5885e-10, cost=1.0587e-19
n=8, max_err=5.7838e-10, cost=6.1755e-19
n=9, max_err=2.5336e-10, cost=9.2695e-20
n=10, max_err=9.1704e-11, cost=1.1776e-20
```

The error is bigger for the first degree, but then it decreases faster than the regular polynomial.



(c)

Here we run the `fit_better()` method to get the following results:



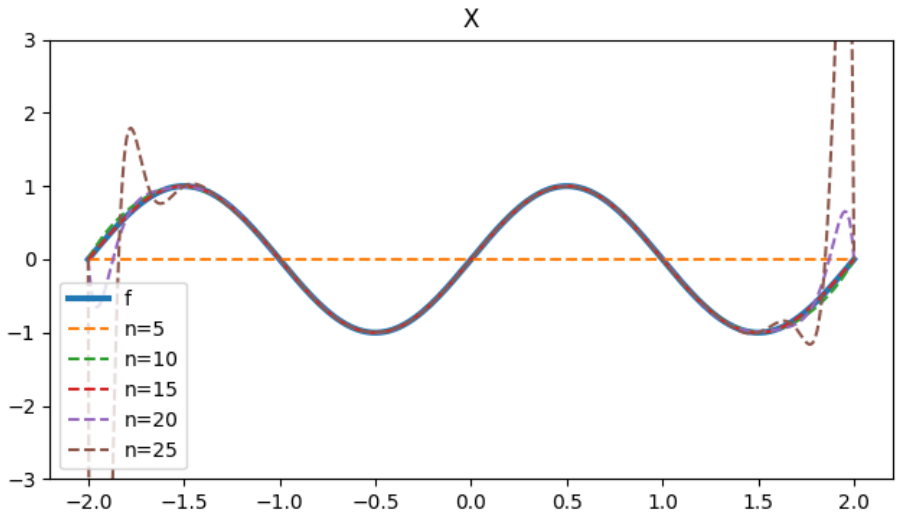

```
max_err=2.6711e-05, cost=1.4724e-09
```

Which is a very nice fit and the error is similar to the error of a 5 degree polynomial. That makes it a better fit, with one less parameter to calculate.

2 *Code is at two.py - at the end you can find method calls for (a), (b) and 3*

a

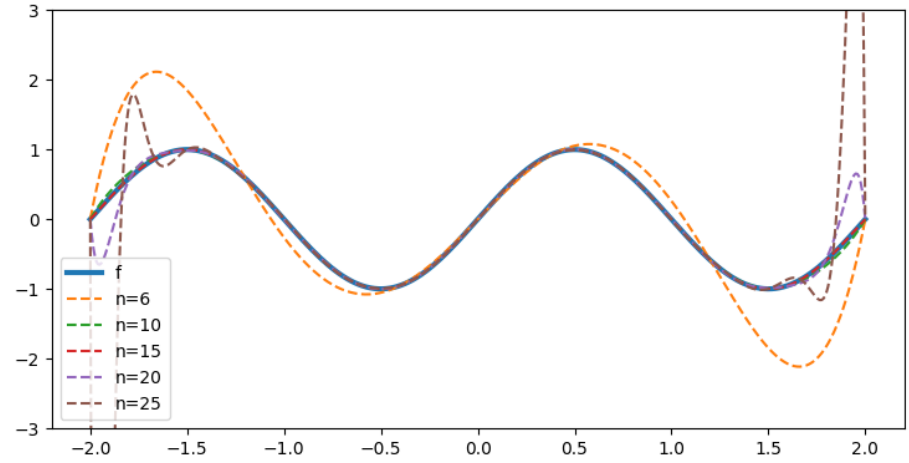
Running fit() with the primitive polynomial we get




```
n=5 err=15.79556899893131
n=10 err=0.5815157875793254
n=15 err=0.03502807033393997
n=20 err=3.769254750738436
n=25 err=48.99896722108643
```

In general the approximation is ok, it diverges at the higher degrees close to the edges.
At `n=5` it is not able to find an answer better than a straight line at the y axis, but with `n=6` it is already able to atleast get something that resembles the target function:

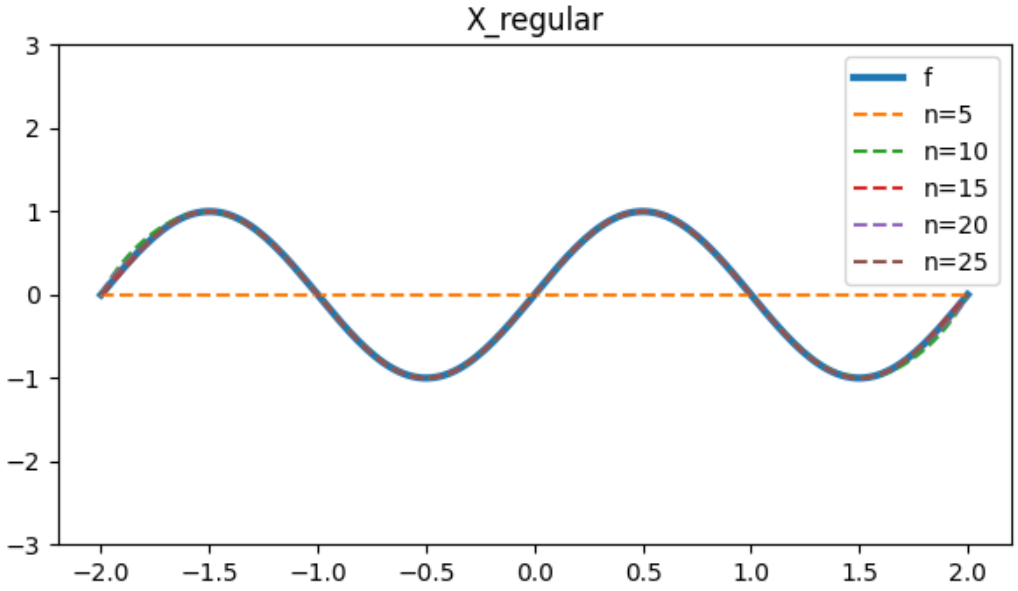
```
n=6 err=12.770771809448993
```



The error is still pretty big and close to the previous 15.79 though.

b

Running `fit(X_reg …)` with the regularized basis it gives:

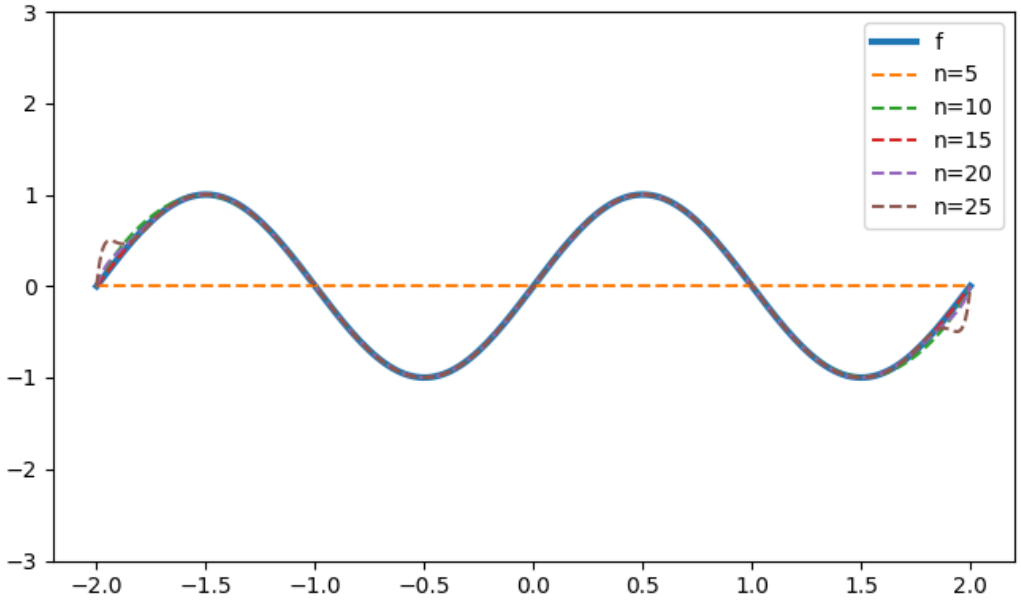



```
n=5 err=15.79556899893131
n=10 err=0.5815157887369966
n=15 err=0.11634396289731887
n=20 err=0.20631967989296487
n=25 err=0.05309501341715034
```

Indeed the errors are much better at the higher degree polynomials.

3 *Code is at two.py. Uncomment the line with* `fit(X_leg..`

Did the mapping by the formula - `f_n(x) = P_n(2 * (x - a) / (b-a) - 1)` to get the following plot and error:




```
n=5 err=15.79556899893131
n=10 err=0.5815157903839132
n=15 err=0.004325502835684117
n=20 err=0.40938088786603055
n=25 err=1.387177674583977
```

The values are indeed much better, the error at n=15 is almost 3 times lower than the regularized basis from 2.b
Also, the higher degree ones don't diverge much at the edges, which makes it much better at n=25 compared to the primitive polynomial basis

4 *Code is at four.py*

Running the script for all three data samples we get a nice fit for the data.
The found a, b for each data file is written in the plot legend.