Exercise 3

main.ipynb — curse_of_dimensionality

main.ipynb > Question 2 and 3 > # We define the parameters as fixed values

Generate | Code | Markdown | Run All | Restart | Clear All Outputs | Variables | Outline

interpolation (Python 3.12.7)

Question 2 and 3

Question 2 and 3 are closely related and are therefore jointly addressed.

2. Evaluate the interpolator on a much finer grid than the original t grid.
3. Show the results of question 2 in a plot, showing both the exact function and the predictions of the interpolator.

```python
# We define the parameters as fixed values
a = 0.1
b = -0.13
c = 9

def get_real_and_interpolated_values(a, b, c, nr_training_t, nr_evaluated_t):
    """
    This function allows us to define on which grid we want to train the model and on which grid we want to evaluate it.

    Args:
        - a, b, c: parameters of the function f(t)
        - nr_training_t: number of points at which we train the model
        - nr_evaluated_t: number of points at which we evaluate the model

    Returns:
        - f_values: values of f(t) for the given parameters
        - f_interpolated_values: values of the interpolated function
        - t2: grid at which we evaluate the interpolated function
    """
    # We create an array of training t's between 0 and 1 (linear space)
    t = np.linspace(0, 1, nr_training_t)

    # We obtain the values of f(t) for the given parameters
    f_values = f(t, a, b, c)

    # We interpolate the values of f(t) using a linear interpolation
    f_interpolated = scipy.interpolate.interp1d(t, f_values, kind='linear')

    # Now evaluate f_interpolated at many more points as given by the number of evaluated t's
    t2 = np.linspace(0, 1, nr_evaluated_t)
    f_interpolated_values = f_interpolated(t2)
    f_values = f(t2, a, b, c)

    return f_values, f_interpolated_values, t2

# First, let's see what happens when we evaluate t at a much lower resolution grid than in the previous exercise.
f_values, f_interpolated_values, t = get_real_and_interpolated_values(a, b, c, 10, 10000) # We choose a training grid a factor 10 smaller than in the previous exercise (evaluation grid of 10000).
plot_against_interpolation(a, b, c, f_interpolated_values, t=t, title="Low Res: Interpolated vs. true values")
plot_ratio(f_values, f_interpolated_values, cutoff=[-10,10], title="Low Res: Ratio between true values and interpolated values (cut off at -10 and 10)")
```

[43]

ex2* ⊗ 0 ⚠ 0 0

main.ipynb — curse_of_dimensionality

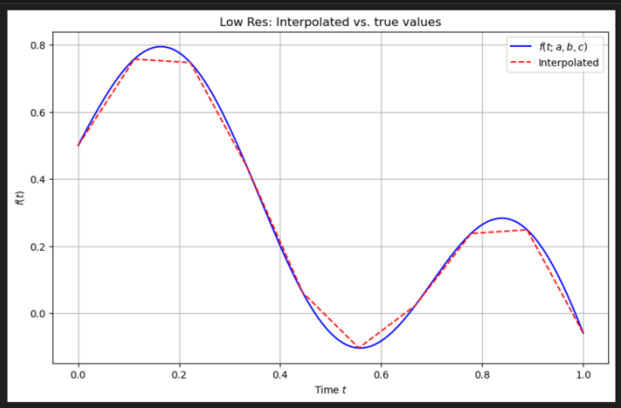
main.ipynb > Question 2 and 3 > # We define the parameters as fixed values

Generate | Code | Markdown | Run All | Restart | Clear All Outputs | Variables | Outline

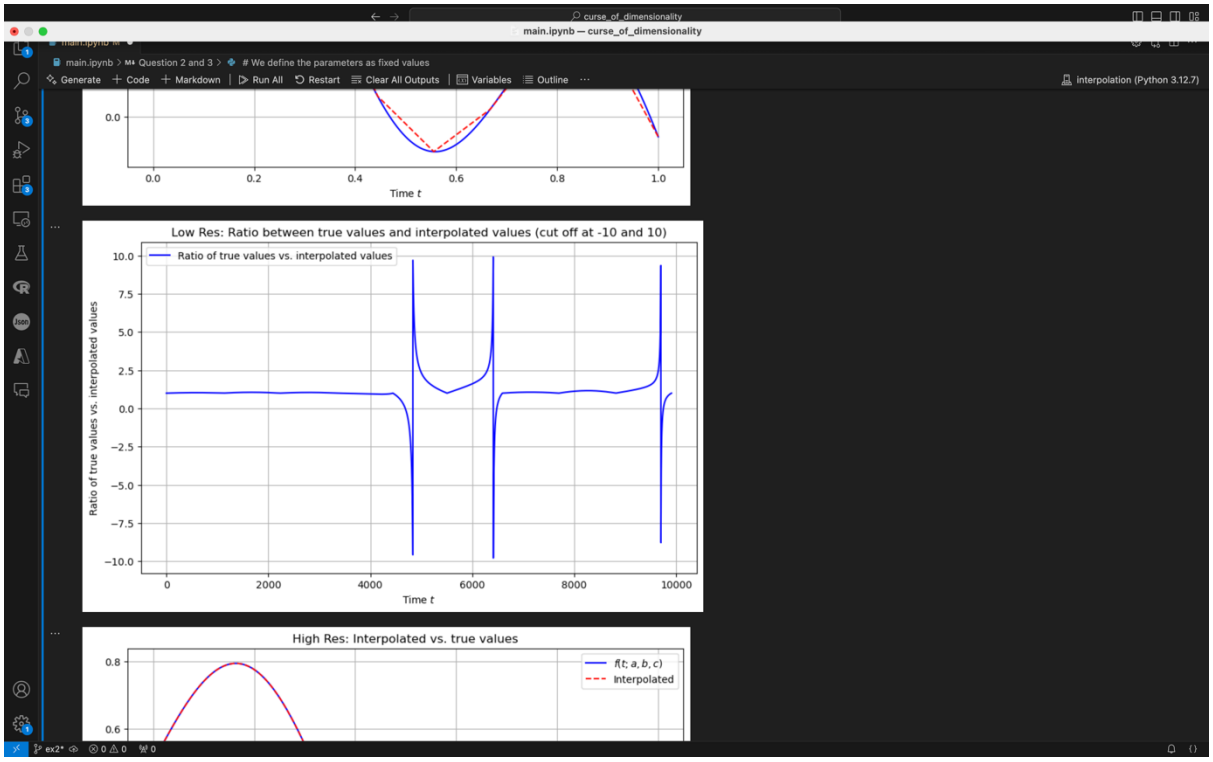
interpolation (Python 3.12.7)

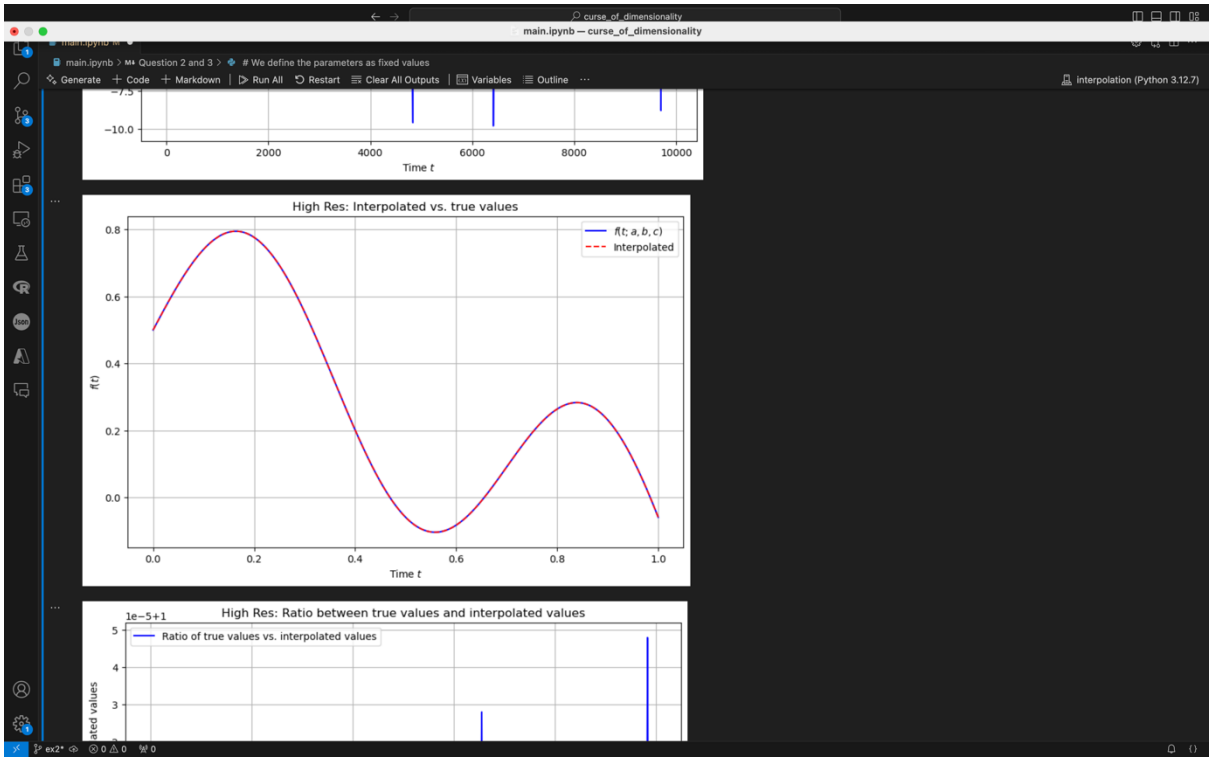
```python
    # Now evaluate f_interpolated at many more points as given by the number of evaluated t's
    t2 = np.linspace(0, 1, nr_evaluated_t)
    f_interpolated_values = f_interpolated(t2)
    f_values = f(t2, a, b, c)

    return f_values, f_interpolated_values, t2

# First, let's see what happens when we evaluate t at a much lower resolution grid than in the previous exercise.
f_values, f_interpolated_values, t = get_real_and_interpolated_values(a, b, c, 10, 10000) # We choose a training grid a factor 10 smaller than in the previous exercise (evaluation grid of 10000).
plot_against_interpolation(a, b, c, f_interpolated_values, t=t, title="Low Res: Interpolated vs. true values")
plot_ratio(f_values, f_interpolated_values, cutoff=[-10,10], title="Low Res: Ratio between true values and interpolated values (cut off at -10 and 10)")

# Now, let's see what happens when we evaluate t at a much higher resolution grid.
f_values, f_interpolated_values, t = get_real_and_interpolated_values(a, b, c, 10000, 1000) # We choose a training grid a factor 10 bigger than in the previous exercise (evaluation grid of 10000).
plot_against_interpolation(a, b, c, f_interpolated_values, t=t, title="High Res: Interpolated vs. true values")
plot_ratio(f_values, f_interpolated_values, cutoff=[-np.inf,np.inf], title="High Res: Ratio between true values and interpolated values")
```

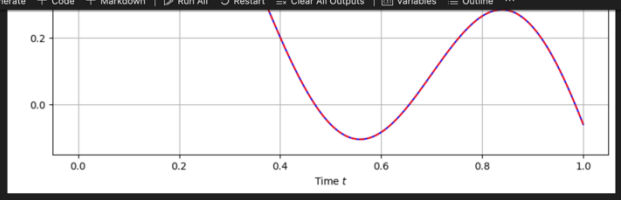
[43] ✓ 0.4s Python

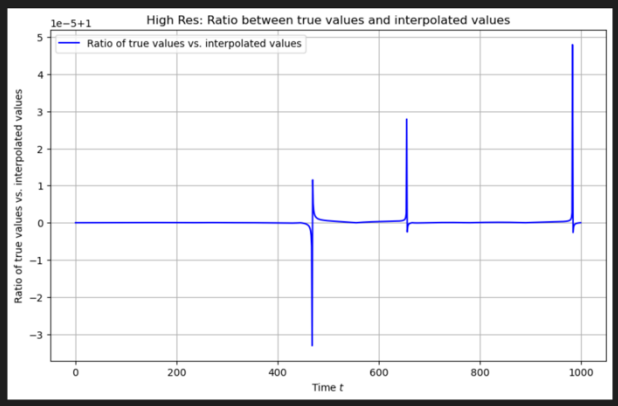


ex2* ⊗ 0 ⚠ 0 0


main.ipynb — curse_of_dimensionality
main.ipynb > Question 2 and 3 > # We define the parameters as fixed values
Generate Code Markdown Run All Restart Clear All Outputs Variables Outline
interpolation (Python 3.12.7)
Time t
Low Res: Ratio between true values and interpolated values (cut off at -10 and 10)
Ratio of true values vs. interpolated values
Ratio of true values vs. interpolated values
Time t
High Res: Interpolated vs. true values
f(t; a, b, c)
Interpolated


main.ipynb — curse_of_dimensionality
main.ipynb > Question 2 and 3 > # We define the parameters as fixed values
Generate Code Markdown Run All Restart Clear All Outputs Variables Outline
interpolation (Python 3.12.7)
Time t
High Res: Interpolated vs. true values
f(t; a, b, c)
Interpolated
f(t)
Time t
High Res: Ratio between true values and interpolated values
1e−5+1
Ratio of true values vs. interpolated values





Conclusion: We may clearly see, that evaluating the interpolated function of more points of our original function yields a much better result, compared to evaluating the interpolation on only a few points.

Exercise 6

PS: I realise that my multidimensional interpolation function isn't truly multidimensional yet (but very much restricted to a and t), however for the sake of brevity this was seen as good enough.

```python
def get_interpolated_and_evaluated_values_multidim(args):
    """
    {
        "interpolated_params": ["a", "t"],
        "set_params": ["b", "c"],
        "a": {
            "min": 0,
            "max": 1,
            "nr_training": 10,
            "nr_evaluated": 100
        },
        "t": {
            "min": 0,
            "max": 1,
            "nr_training": 100,
            "nr_evaluated": 1000
        },
        "b": {
            "min": -0.13,
            "max": 0.13,
            "nr_training": 1,
            "nr_evaluated": 1
        },
        "c": {
            "min": 9,
            "max": 9,
            "nr_training": 1,
            "nr_evaluated": 1
        }
    }
    """
    # Get the training linespaces for the parameters
    training_linespaces = {}
    for param in args:
        training_linespaces[param] = np.linspace(args[param]["min"], args[param]["max"], args[param]["nr_training"])

    # # Get the evaluated linespaces for the parameters
    evaluated_linespaces = {}
    for param in args:
        evaluated_linespaces[param] = np.linspace(args[param]["min"], args[param]["max"], args[param]["nr_evaluated"])

    # Get interpolated values for the training linespaces
    f_training_values, grid_training_params = get_interpolated_values(training_linespaces)
    f_evaluated_values, grid_evaluated_params = get_interpolated_values(evaluated_linespaces)

    # Create a 2D interpolator for `a` and `t` using training values
    a_training = grid_training_params["a"].flatten()
    t_training = grid_training_params["t"].flatten()
    f_training_flat = f_training_values.flatten()
```

```python
from scipy.interpolate import LinearNDInterpolator

def get_interpolated_values(args):
    """
    {
        "a": linespace_a,
        "t": linespace_t,
        "b": linespace_b,
        "c": linespace_c
    }
    """
    # Create meshgrid for all parameters
    grids = np.meshgrid(*args.values(), indexing="ij")

    # Create a dictionary mapping parameter names to their grids
    grid_params = {param: grid for param, grid in zip(args.keys(), grids)}

    f_values = f(
        grid_params["t"],
        grid_params["a"],
        grid_params["b"],
        grid_params["c"]
    )

    return f_values, grid_params


def get_interpolated_and_evaluated_values_multidim(args):
    """
    {
        "interpolated_params": ["a", "t"],
        "set_params": ["b", "c"],
        "a": {
            "min": 0,
            "max": 1,
            "nr_training": 10,
            "nr_evaluated": 100
        },
        "t": {
```

6 Question

Create an interpolator that interpolates over a (same range as previous question) and returns the full time series (i.e., values of f for all time points) over the original time grid, i.e., t.

Plot the result for a = 0.123, like in question 3.

Plot the ratio, like in question 4.

main.ipynb — curse_of_dimensionality

main.ipynb > 6 Question > from scipy.interpolate import LinearNDInterpolator

Generate | Code | Markdown | Run All | Restart | Clear All Outputs | Variables | Outline

interpolation (Python 3.12.7)

```python
args = {
        "a": {
            "min": 0,
            "max": 1,
            "nr_training": 10,
            "nr_evaluated": 100
        },
        "t": {
            "min": 0,
            "max": 1,
            "nr_training": 100,
            "nr_evaluated": 1000
        },
        "b": {
            "min": -0.13,
            "max": 0.13,
            "nr_training": 1,
            "nr_evaluated": 1
        },
        "c": {
            "min": 9,
            "max": 9,
            "nr_training": 1,
            "nr_evaluated": 1
        }
    }

f_interpolated, f_training_values, f_evaluated_values, (a_grid, t_grid) = get_interpolated_and_evaluated_values_multidim(args)
print("Shape of f_training_values:", f_training_values.shape)
print("Shape of f_evaluated_values:", f_evaluated_values.shape)
# print("Shape of f_values:", f_values.shape)

# Define the target value for a
target_a = 0.125

# Find the index in a_grid where the value is closest to target_a
a_index = np.abs(a_grid[:, 0] - target_a).argmin()

# Extract the corresponding t values and f_interpolated values
t_values_at_target_a = t_grid[a_index, :]
f_values_at_target_a = f_interpolated[a_index, :]

# Print the results
print(f"Interpolated values for a = {target_a}:")
print("t values:", t_values_at_target_a)
print("f values:", f_values_at_target_a)
plot_two_functions(f(t_values_at_target_a, target_a, b, c), f_values_at_target_a, title="Interpolated vs True Values at a=0.125")
plot_ratio(f(t_values_at_target_a, target_a, b, c), f_values_at_target_a, title="Ratio between true values and interpolated values")
```

[65] ✓ 0.2s Python

ex2* ⊗ 0 ⚠ 2 0

main.ipynb — curse_of_dimensionality

main.ipynb > 6 Question > from scipy.interpolate import LinearNDInterpolator

Generate | Code | Markdown | Run All | Restart | Clear All Outputs | Variables | Outline

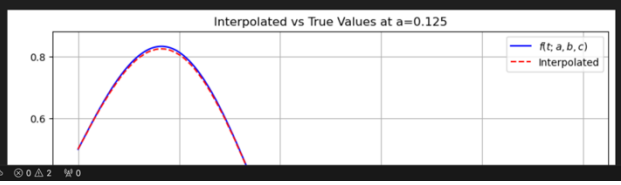
interpolation (Python 3.12.7)

```python
print("t values:", t_values_at_target_a)
print("f values:", f_values_at_target_a)
plot_two_functions(f(t_values_at_target_a, target_a, b, c), f_values_at_target_a, title="Interpolated vs True Values at a=0.125")
plot_ratio(f(t_values_at_target_a, target_a, b, c), f_values_at_target_a, title="Ratio between true values and interpolated values")
```

[65] ✓ 0.2s Python

```
Shape of f_training_values: (10, 100, 1, 1)
Shape of f_evaluated_values: (100, 1000, 1, 1)
Interpolated values for a = 0.125:
t values: [0.         0.001001   0.002002   0.003003   0.004004   0.00500501
 0.00600601 0.00700701 0.00800801 0.00900901 0.01001001 0.01101101
 0.01201201 0.01301301 0.01401401 0.01501502 0.01601602 0.01701702
 0.01801802 0.01901902 0.02002002 0.02102102 0.02202202 0.02302302
 0.02402402 0.02502503 0.02602603 0.02702703 0.02802803 0.02902903
 0.03003003 0.03103103 0.03203203 0.03303303 0.03403403 0.03503504
 0.03603604 0.03703704 0.03803804 0.03903904 0.04004004 0.04104104
 0.04204204 0.04304304 0.04404404 0.04504505 0.04604605 0.04704705
 0.04804805 0.04904905 0.05005005 0.05105105 0.05205205 0.05305305
 0.05405405 0.05505506 0.05605606 0.05705706 0.05805806 0.05905906
 0.06006006 0.06106106 0.06206206 0.06306306 0.06406406 0.06506507
 0.06606607 0.06706707 0.06806807 0.06906907 0.07007007 0.07107107
 0.07207207 0.07307307 0.07407407 0.07507508 0.07607608 0.07707708
 0.07807808 0.07907908 0.08008008 0.08108108 0.08208208 0.08308308
 0.08408408 0.08508509 0.08608609 0.08708709 0.08808809 0.08908909
 0.09009009 0.09109109 0.09209209 0.09309309 0.09409409 0.0950951
 0.0960961  0.0970971  0.0980981  0.0990991  0.1001001  0.1011011
 0.1021021  0.1031031  0.1041041  0.10510511 0.10610611 0.10710711
 0.10810811 0.10910911 0.11011011 0.11111111 0.11211211 0.11311311
 0.11411411 0.11511512 0.11611612 0.11711712 0.11811812 0.11911912
 0.12012012 0.12112112 0.12212212 0.12312312 0.12412412 0.12512513
 0.12612613 0.12712713 0.12812813 0.12912913 0.13013013 0.13113113
...
  1.37259077e-02  9.94095688e-03  6.15600603e-03  2.37105517e-03
 -1.41389568e-03 -5.32657649e-03 -1.02992151e-02 -1.42368933e-02
 -1.81745715e-02 -2.21122497e-02 -2.60499278e-02 -2.99876060e-02
 -3.39252842e-02 -3.78629624e-02 -4.18006405e-02 -4.57383187e-02]
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...



ex2* ⊗ 0 ⚠ 2 0

curse_of_dimensionality

main.ipynb — curse_of_dimensionality

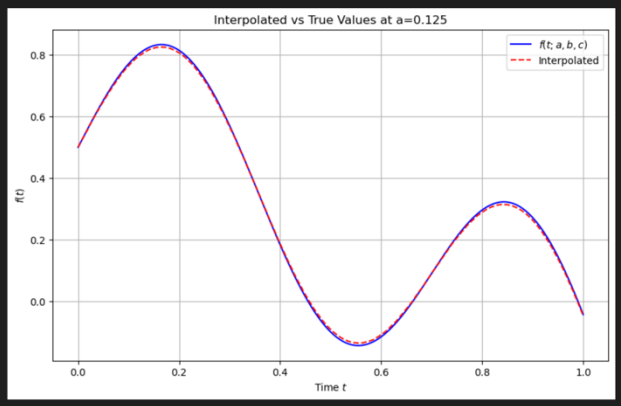
main.ipynb > 6 Question > from scipy.interpolate import LinearNDInterpolator

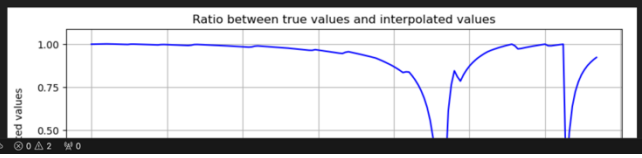
Generate | Code | Markdown | Run All | Restart | Clear All Outputs | Variables | Outline

interpolation (Python 3.12.7)

```
 0.12012012 0.12112112 0.12212212 0.12312312 0.12412412 0.12512513
 0.12612613 0.12712713 0.12812813 0.12912913 0.13013013 0.13113113
 ...
  1.37259077e-02  9.94095688e-03  6.15600603e-03  2.37105517e-03
 -1.41389568e-03 -5.32657649e-03 -1.02992151e-02 -1.42368933e-02
 -1.81745715e-02 -2.21122497e-02 -2.60499278e-02 -2.99876060e-02
 -3.39252842e-02 -3.78629624e-02 -4.18006405e-02 -4.57383187e-02]
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...





ex2* ⊗ 0 ⚠ 2 0

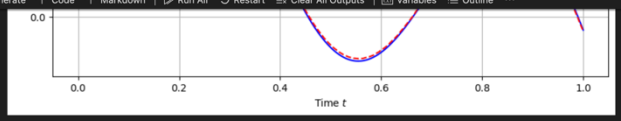
curse_of_dimensionality

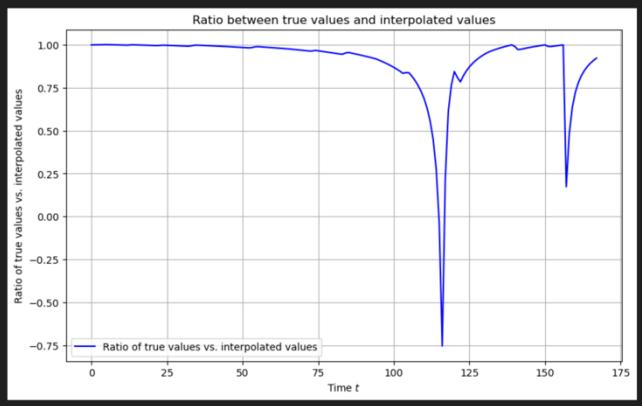
main.ipynb — curse_of_dimensionality

main.ipynb > 6 Question > from scipy.interpolate import LinearNDInterpolator

Generate | Code | Markdown | Run All | Restart | Clear All Outputs | Variables | Outline

interpolation (Python 3.12.7)






```
t = np.linspace(0, 1, 100)
a = np.linspace(0, 1, 10)
values = np.array([[f(t_val, a_val, b, c) for t_val in t] for a_val in a])

# Create the interpolator
f_interpolated = scipy.interpolate.RegularGridInterpolator((a, t), values)
```

ex2* ⊗ 0 ⚠ 2 0

curse_of_dimensionality

main.ipynb — curse_of_dimensionality

main.ipynb > 6 Question > from scipy.interpolate import LinearNDInterpolator

Generate | Code | Markdown | Run All | Restart | Clear All Outputs | Variables | Outline

interpolation (Python 3.12.7)

```python
            "max": 1,
            "nr_training": 100,
            "nr_evaluated": 1000
        },
        "b": {
            "min": -0.13,
            "max": 0.13,
            "nr_training": 1,
            "nr_evaluated": 1
        },
        "c": {
            "min": 9,
            "max": 9,
            "nr_training": 1,
            "nr_evaluated": 1
        }
    }
    """
    # Get the training linespaces for the parameters
    training_linespaces = {}
    for param in args:
        training_linespaces[param] = np.linspace(args[param]["min"], args[param]["max"], args[param]["nr_training"])

    # # Get the evaluated linespaces for the parameters
    evaluated_linespaces = {}
    for param in args:
        evaluated_linespaces[param] = np.linspace(args[param]["min"], args[param]["max"], args[param]["nr_evaluated"])

    # Get interpolated values for the training linespaces
    f_training_values, grid_training_params = get_interpolated_values(training_linespaces)
    f_evaluated_values, grid_evaluated_params = get_interpolated_values(evaluated_linespaces)

    # Create a 2D interpolator for `a` and `t` using training values
    a_training = grid_training_params["a"].flatten()
    t_training = grid_training_params["t"].flatten()
    f_training_flat = f_training_values.flatten()
    interpolator = LinearNDInterpolator(list(zip(a_training, t_training)), f_training_flat)

    # Generate a 2D grid for `a` and `t` using evaluated linespaces
    a_eval = evaluated_linespaces["a"]
    t_eval = evaluated_linespaces["t"]
    a_grid, t_grid = np.meshgrid(a_eval, t_eval, indexing="ij")


    # Interpolate on the evaluated grid
    f_interpolated = interpolator(a_grid, t_grid)

    return f_interpolated, f_training_values, f_evaluated_values, (a_grid, t_grid)
```

[65]

ex2* ⊗ 0 ⚠ 2 0

Exercise 11

main.ipynb — curse_of_dimensionality

main.ipynb > Question 12

Generate | Code | Markdown | Run All | Restart | Clear All Outputs | Variables | Outline ···

interpolation (Python 3.12.7)

Question 11

Create the interpolator over the parameter space (a, b) interpolating over samples of the function evaluated at the original time grid t.

As you will realise, we are dealing with an irregular grid and need the `griddata` method.

```python
parameter_space = sample_scaled.copy() # Data obtained from Latin Hypercube Sampling
print(parameter_space)

def convert_parameter_space(parameter_space):
    t_values = np.linspace(0, 1, len(parameter_space))
    return [(a, b, t) for (a, b), t in zip(parameter_space, t_values)]

conv_parameter_space = convert_parameter_space(parameter_space)

# Fixed value of c
c_fixed = 0.9

t = np.linspace(0, 1, len(parameter_space))
# Calculate f(t, a, b, c_fixed) for each point in parameter_space
f_vals = [f(t, a, b, c=c_fixed) for a, b, t in conv_parameter_space]

def interpolate_at_points(parameter_space):
    # Extract a, b, t values from parameter_space
    a_vals, b_vals, t_vals = zip(*parameter_space)

    # Perform interpolation using only the points in parameter_space
    interpolated_values = scipy.interpolate.griddata(
        points=(a_vals, b_vals, t_vals),
        values=f_vals,
        xi=parameter_space,
        method='linear'
    )

    return interpolated_values

interpolated_values = interpolate_at_points(conv_parameter_space)
print(interpolated_values)
```

[85] ✓ 0.0s Python

```
[[ 9.57291650e-01 -2.05443899e-01]
 [ 9.41036135e-02 -1.46152588e-02]
 [ 9.45014247e-01 -8.23964812e-02]
 [ 3.42658497e-01 -4.76916852e-01]
```

ex2* ⊗ 0 ⚠ 2 ⚑ 0

Exercise 13

main.ipynb — curse_of_dimensionality

main.ipynb > Exercise 13 > import timeit

Generate + Code + Markdown | Run All Restart Clear All Outputs | Variables Outline ···

interpolation (Python 3.12.7)

Exercise 13

Compare memory and time of (i) the original function call and (ii) the interpolator call.

```python
import timeit
import tracemalloc
import numpy as np

def compare_functions(func_a, func_b, *args, n_trials=1000):
    """
    Compares the execution time and memory usage of two functions.

    Parameters:
    - func_a: First function to compare.
    - func_b: Second function to compare.
    - *args: Arguments to be passed to both functions.
    - n_trials: Number of times to execute the functions for timing (default=1000).

    Returns:
    - A dictionary with time and memory stats for both functions.
    """
    def measure_time_and_memory(func, *args):
        # Measure memory usage
        tracemalloc.start()
        start_snapshot = tracemalloc.take_snapshot()

        # Measure execution time
        time_start = timeit.default_timer()
        func(*args)
        time_end = timeit.default_timer()

        # Capture memory usage
        end_snapshot = tracemalloc.take_snapshot()
        tracemalloc.stop()

        # Calculate memory difference
        memory_stats = end_snapshot.compare_to(start_snapshot, 'lineno')
        total_memory = sum(stat.size_diff for stat in memory_stats)

        return time_end - time_start, total_memory

    # Measure function A
    time_a = timeit.timeit(lambda: func_a(*args), number=n_trials) / n_trials
    memory_a, _ = measure_time_and_memory(func_a, *args)

    # Measure function B
```

[98]

ex2* ⊗ 0 ⚠ 2 ⚑ 0

main.ipynb — curse_of_dimensionality

main.ipynb > Exercise 13 > import timeit

Generate + Code + Markdown | Run All Restart Clear All Outputs | Variables Outline ···

interpolation (Python 3.12.7)

```python
        # Capture memory usage
        end_snapshot = tracemalloc.take_snapshot()
        tracemalloc.stop()

        # Calculate memory difference
        memory_stats = end_snapshot.compare_to(start_snapshot, 'lineno')
        total_memory = sum(stat.size_diff for stat in memory_stats)

        return time_end - time_start, total_memory

    # Measure function A
    time_a = timeit.timeit(lambda: func_a(*args), number=n_trials) / n_trials
    memory_a, _ = measure_time_and_memory(func_a, *args)

    # Measure function B
    time_b = timeit.timeit(lambda: func_b(*args), number=n_trials) / n_trials
    memory_b, _ = measure_time_and_memory(func_b, *args)

    # Compile results
    results = {
        'Original Function (Func A)': {
            'Average Time (s)': time_a,
            'Memory Usage (bytes)': memory_a,
        },
        'Interpolator Function (Func B)': {
            'Average Time (s)': time_b,
            'Memory Usage (bytes)': memory_b,
        }
    }

    return results


# Example usage
def function_a(t):
    return f(t, 0.5, -0.13, 9)

def function_b(x):
    return f_interpolated(t)

# We create an array of training t's between 0 and 1 (linear space)
t = np.linspace(0, 1, 10000)

results = compare_functions(function_a, function_b, t, n_trials=1000)
print(results)

# Make a bar chart to show the results of the comparison
def plot_comparison(results):
    # Extract the function names and their stats
    function_names = list(results.keys())
```

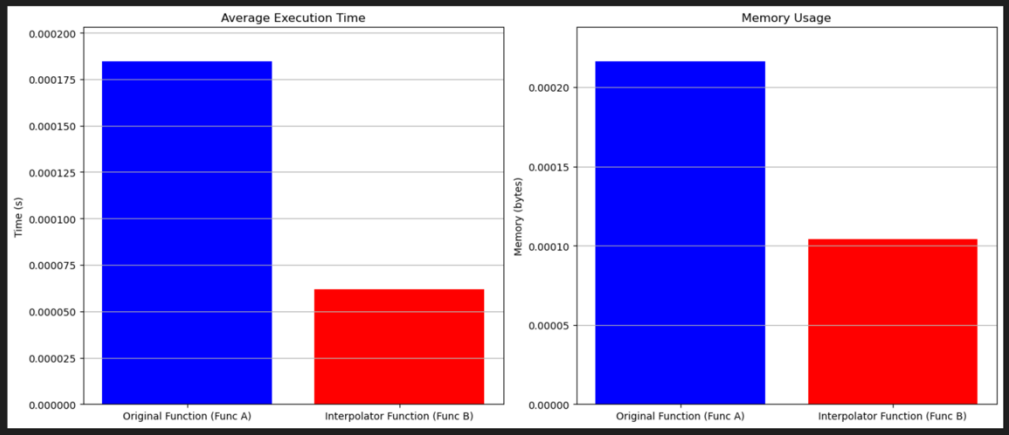
[98]

ex2* ⊗ 0 ⚠ 2 ⚑ 0

```
plt.show()

plot_comparison(results)
```

[98] ✓ 0.4s Python

```
{'Original Function (Func A)': {'Average Time (s)': 0.000184553416998824 1, 'Memory Usage (bytes)': 0.00021620799088850617}, 'Interpolator Function (Func B)': {'Average Time (s)': 6.180008299998008e-05, 'Memory Usage
```



Comment

The interpolator function is faster and more memory efficient to call than the original function. This may be explained by the fact that the interpolator works on a collection of linear functions, while the original function needs to make more complex sine/cosine etc. computations. Therefore, once we have estimated our interpolator, it is more efficient to use the interpolator than the original function.