Programming for Engineers

Lecture 5 - Array processing

Table of contents

Lecture 5 - Array processing **1**

Sorting . . . 1

Sorting algorithms . . . 3

Comparison of sorting algorithms . . . 14

Sliding window operations . . . 17

Fixed-size window . . . 17

Variable-size window . . . 20

Convolution . . . 22

1D convolution . . . 23

2D convolution . . . 26

Sparse matrices . . . 33

Coordinate list (COO) . . . 34

Operations on COO sparse matrices . . . 37

Compressed sparse rows (CSR) . . . 40

Lecture 5 - Array processing

```
import numpy as np
```

Sorting

Sorting is one of the most common operations on arrays. There are many approaches to sorting - each performing differently in different situations.

Preparation of the visualization code

Before we start with the explanations of the sorting algorithms, we should create some code that will allow us to test and visualize what is actually happening.

We will be printing states of the array that is being sorted after each step of the algorithm. Note that an **algorithm having fewer steps does not mean more efficient algorithm**. For **some sorting algorithms, a single step might take more operations** (such as looping through the entire array for selection sort).

In the following sections, the sorting functions will have `yield` statements in them. These are for visualization of the sorting process. You can ignore them when reading the code. The `yield` returns the given value (like `return`) but *does* ***not*** *exit* the function.

To show the sorting algorithms in action, let's create a function that will make an unsorted array (probably) and run the sorting algorithm on it.

```
def pretty_format_array(row, digits=2):
    return "[" + ', '.join(f"{x:>{digits}d}" for x in row) + "]"

def test_sort(sorting_func, show_full=False):
    np.random.seed(42)
    a = np.random.randint(0, 20, size=10)
    if show_full:
        print("Unsorted array:")
        print(pretty_format_array(a))
    states = []
    for i, state in enumerate(sorting_func(a)):
        if i == 0 and not show_full:
            print("Unsorted array:")
        # elif i == 1:
        #     print("Sorting process:")
        print(f"{i:2d}:", pretty_format_array(state))
        states.append(state)
    return states
```

To make the visualization nice, let's plot the sorting process and make an animation out of it. Don't worry about the details of the animation code - you don't need to understand it. The animation will show a bar graph, where each *bar* represents an element in the array. The height of the bar is the value of the element.

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
```

```
def animate_sort(states, title=""):
    numel = len(states[-1])
    for i, state in enumerate(states):
        states[i] = np.pad(
            state, (0, numel - len(state)),
            mode="constant", constant_values=0)
    fig, ax = plt.subplots()
    bar_rects = ax.bar(range(numel), states[0], align="edge")
    ax.set_title(title)
    ax.set_xlim(0, numel)
    ax.set_ylim(0, max(max(state) for state in states) * 1.1)

    text = ax.text(0.02, 0.95, "", transform=ax.transAxes)

    def update_plot(frame):
        for rect, val in zip(bar_rects, states[frame]):
            rect.set_height(val)
        text.set_text(f"Step {frame}")
        return bar_rects

    ani = animation.FuncAnimation(fig, update_plot, frames=len(states),
                                  interval=300, repeat=False)
    plt.close()
    return HTML(ani.to_html5_video())
```

Sorting algorithms

Bubble Sort

Bubble sort is one of the simplest sorting algorithms - simply take two neighboring elements and swap them if they are out of order or leave them unchanged if they are in order. Do this for each pair of elements in the array. Repeat until the array is sorted. This has the effect of large values "bubbling" to the top of the array, hence the name.

While simple in principle, for N elements, each "swapping" pass (comparing every pair of elements once) requires N comparisons. In worst case, i.e., *reversed* array, each value would need to be swapped with each other value. Therefore, we would need N passes and thus the total number of comparisons would be N^2. Actually, (slightly) less, since the largest value would "bubble" to the top of the array and thus we don't need to "touch" the last index on the next pass; likewise for the second largest value, etc. One small advantage is that the sorting is performed in-place and thus requires (virtually) no additional memory.

Here is an example implementation of the algorithm:

```
def bubble_sort(a):
    n = len(a)
    yield a.copy()  # Initial state
    for i in range(n):
        for j in range(0, n - i - 1):
            if a[j] > a[j + 1]:  # compare the neighboring items
                a[j], a[j + 1] = a[j + 1], a[j]  # swap the items (if necessary)
                yield a.copy()  # Yield state after each swap
    return a
```

Now, let's test it:

```
states = test_sort(bubble_sort)
```

```
Unsorted array:
 0: [ 6, 19, 14, 10,  7,  6, 18, 10, 10,  3]
 1: [ 6, 14, 19, 10,  7,  6, 18, 10, 10,  3]
 2: [ 6, 14, 10, 19,  7,  6, 18, 10, 10,  3]
 3: [ 6, 14, 10,  7, 19,  6, 18, 10, 10,  3]
 4: [ 6, 14, 10,  7,  6, 19, 18, 10, 10,  3]
 5: [ 6, 14, 10,  7,  6, 18, 19, 10, 10,  3]
 6: [ 6, 14, 10,  7,  6, 18, 10, 19, 10,  3]
 7: [ 6, 14, 10,  7,  6, 18, 10, 10, 19,  3]
 8: [ 6, 14, 10,  7,  6, 18, 10, 10,  3, 19]
 9: [ 6, 10, 14,  7,  6, 18, 10, 10,  3, 19]
10: [ 6, 10,  7, 14,  6, 18, 10, 10,  3, 19]
11: [ 6, 10,  7,  6, 14, 18, 10, 10,  3, 19]
12: [ 6, 10,  7,  6, 14, 10, 18, 10,  3, 19]
13: [ 6, 10,  7,  6, 14, 10, 10, 18,  3, 19]
14: [ 6, 10,  7,  6, 14, 10, 10,  3, 18, 19]
15: [ 6,  7, 10,  6, 14, 10, 10,  3, 18, 19]
16: [ 6,  7,  6, 10, 14, 10, 10,  3, 18, 19]
17: [ 6,  7,  6, 10, 10, 14, 10,  3, 18, 19]
18: [ 6,  7,  6, 10, 10, 10, 14,  3, 18, 19]
19: [ 6,  7,  6, 10, 10, 10,  3, 14, 18, 19]
20: [ 6,  6,  7, 10, 10, 10,  3, 14, 18, 19]
21: [ 6,  6,  7, 10, 10,  3, 10, 14, 18, 19]
22: [ 6,  6,  7, 10,  3, 10, 10, 14, 18, 19]
23: [ 6,  6,  7,  3, 10, 10, 10, 14, 18, 19]
24: [ 6,  6,  3,  7, 10, 10, 10, 14, 18, 19]
```

```
25: [ 6,  3,  6,  7, 10, 10, 10, 14, 18, 19]
26: [ 3,  6,  6,  7, 10, 10, 10, 14, 18, 19]
```

Selection Sort

Selection sort, like the Bubble sort, isn't, in most cases, practical. It loops over the array, finds the smallest value and puts it to the first position in the array (where it should be, when the array is sorted). Then it advances to the second position in the array, finds the second smallest value and puts it there. This way, it progressively "leaves behind" a sorted array. It is similar to the Bubble sort "in reverse" (Bubble sort progressively puts highest values at the end). Because for each element, it needs to loop through the whole (remaining) array, the asymptotic complexity is also $O(n^2)$.

```python
def selection_sort(a):
    yield a.copy()
    n = len(a)
    for i in range(n):
        min_idx = i  # index of the "currently smallest" element
        for j in range(i + 1, n):
            if a[j] < a[min_idx]:  # if an element is smaller
                min_idx = j  # store its index
        # at the end, put the smallest element at the current position
        a[i], a[min_idx] = a[min_idx], a[i]
        yield a.copy()
    return a
```

```python
states = test_sort(selection_sort)
```

```
Unsorted array:
 0: [ 6, 19, 14, 10,  7,  6, 18, 10, 10,  3]
 1: [ 3, 19, 14, 10,  7,  6, 18, 10, 10,  6]
 2: [ 3,  6, 14, 10,  7, 19, 18, 10, 10,  6]
 3: [ 3,  6,  6, 10,  7, 19, 18, 10, 10, 14]
 4: [ 3,  6,  6,  7, 10, 19, 18, 10, 10, 14]
 5: [ 3,  6,  6,  7, 10, 19, 18, 10, 10, 14]
 6: [ 3,  6,  6,  7, 10, 10, 18, 19, 10, 14]
 7: [ 3,  6,  6,  7, 10, 10, 10, 19, 18, 14]
 8: [ 3,  6,  6,  7, 10, 10, 10, 14, 18, 19]
 9: [ 3,  6,  6,  7, 10, 10, 10, 14, 18, 19]
10: [ 3,  6,  6,  7, 10, 10, 10, 14, 18, 19]
```

Insertion Sort

Insertion sort loops through the array (starting from the second position). First, the algorithm stores the current item at position i as *key*. Then, for each previous element at position j, it checks whether the *key* item is smaller. If so, it swaps the items, effectively **moving the smaller item backwards**. If the *key* is larger than the item at j (or we are at the beginning of the array), it puts the *key* item just after j. It essentially moves each item to the *"correct"* position. While the worst case performance is $O(n^2)$, it performs slightly better than the Bubble sort and also does not require any extra memory. It can be **useful for nearly sorted arrays**. For example, adding new items to an already sorted data set (the newly added sample set must be small relative to the whole dataset).

```
def insertion_sort(a):
    yield a.copy()
    for i, key in enumerate(a[1:], start=1):  # take the current element
        j = i - 1  # start with the previous element
        while j >= 0 and a[j] > key:  # while prev. is larger
            a[j + 1] = a[j]  # swap the elements
            j -= 1  # and move to the previous position
            yield a.copy()
        a[j + 1] = key  # finally, (when prev is smaller) put the key at prev. + 1
        yield a.copy()
    return a
```

```
states = test_sort(insertion_sort)
```

```
Unsorted array:
 0: [ 6, 19, 14, 10,  7,  6, 18, 10, 10,  3]
 1: [ 6, 19, 14, 10,  7,  6, 18, 10, 10,  3]
 2: [ 6, 19, 19, 10,  7,  6, 18, 10, 10,  3]
 3: [ 6, 14, 19, 10,  7,  6, 18, 10, 10,  3]
 4: [ 6, 14, 19, 19,  7,  6, 18, 10, 10,  3]
 5: [ 6, 14, 14, 19,  7,  6, 18, 10, 10,  3]
 6: [ 6, 10, 14, 19,  7,  6, 18, 10, 10,  3]
 7: [ 6, 10, 14, 19, 19,  6, 18, 10, 10,  3]
 8: [ 6, 10, 14, 14, 19,  6, 18, 10, 10,  3]
 9: [ 6, 10, 10, 14, 19,  6, 18, 10, 10,  3]
10: [ 6,  7, 10, 14, 19,  6, 18, 10, 10,  3]
11: [ 6,  7, 10, 14, 19, 19, 18, 10, 10,  3]
12: [ 6,  7, 10, 14, 14, 19, 18, 10, 10,  3]
13: [ 6,  7, 10, 10, 14, 19, 18, 10, 10,  3]
14: [ 6,  7,  7, 10, 14, 19, 18, 10, 10,  3]
```

```
15: [ 6,  6,  7, 10, 14, 19, 18, 10, 10,  3]
16: [ 6,  6,  7, 10, 14, 19, 19, 10, 10,  3]
17: [ 6,  6,  7, 10, 14, 18, 19, 10, 10,  3]
18: [ 6,  6,  7, 10, 14, 18, 19, 19, 10,  3]
19: [ 6,  6,  7, 10, 14, 18, 18, 19, 10,  3]
20: [ 6,  6,  7, 10, 14, 14, 18, 19, 10,  3]
21: [ 6,  6,  7, 10, 10, 14, 18, 19, 10,  3]
22: [ 6,  6,  7, 10, 10, 14, 18, 19, 19,  3]
23: [ 6,  6,  7, 10, 10, 14, 18, 18, 19,  3]
24: [ 6,  6,  7, 10, 10, 14, 14, 18, 19,  3]
25: [ 6,  6,  7, 10, 10, 10, 14, 18, 19,  3]
26: [ 6,  6,  7, 10, 10, 10, 14, 18, 19, 19]
27: [ 6,  6,  7, 10, 10, 10, 14, 18, 18, 19]
28: [ 6,  6,  7, 10, 10, 10, 14, 14, 18, 19]
29: [ 6,  6,  7, 10, 10, 10, 10, 14, 18, 19]
30: [ 6,  6,  7, 10, 10, 10, 10, 14, 18, 19]
31: [ 6,  6,  7, 10, 10, 10, 10, 14, 18, 19]
32: [ 6,  6,  7,  7, 10, 10, 10, 14, 18, 19]
33: [ 6,  6,  6,  7, 10, 10, 10, 14, 18, 19]
34: [ 6,  6,  6,  7, 10, 10, 10, 14, 18, 19]
35: [ 3,  6,  6,  7, 10, 10, 10, 14, 18, 19]
```

Merge Sort

Merge sort is a divide-and-conquer algorithm - it recursively divides the array into smaller chunks, merge-sorts these chunks (which means, it also divides them) and then **merge** the sorted chunks to create a sorted array. That is, at each step, it takes the array, splits it (roughly) in half. Then, it splits each half into two more halves, and so on, until each half contains only one (or zero) item. Single (or zero) element array is always sorted - no further processing needed. So, the sorting function will return this "array". When two such arrays are returned (each single element and thus sorted), they are merged - either `left_half + right_half` if the left half is smaller, or `right_half + left_half` if the right half is smaller.

This is then repeated for larger chunks. However, for these, the merging needs to be done element-by-element because, even though each chunk is sorted, it is not guaranteed that elements will be sorted when simply concatenating them. For example, consider the following two arrays (chunks): `a = [1, 5, 9]` and `b = [3, 5, 6, 10]`. They are both individually sorted but you can see we can't simply stack them together to get one sorted array. Rather, merge sort goes item by item for both arrays and puts the lowest item (from `a` and `b`) in the resulting array.

Unlike the previous algorithms, merge sort has a time complexity of $O(n \log n)$, which is better than the quadratic one. However, it requires additional memory - the recursion requires mem-

ory to keep track of the function call (non-recursive version would still need state variables). Still, the time complexity is stable across varying conditions (size, type of data, etc.) - best and worst cases have the same $O(n \log n)$ complexity.

```
def merge(left, right):  # subroutine to merge the halves
    merged = []  # resulting merged array
    left_index, left_length = 0, len(left)  # vars to keep track of the left half
    right_index, right_length = 0, len(right)  # vars to keep track of the right half
    # while both halves are not empty
    while left_index < left_length and right_index < right_length:
        if left[left_index] <= right[right_index]:  # if left is smaller
            merged.append(left[left_index])  # put it in the result
            left_index += 1  # advance left index
        else:  # if right is smaller
            merged.append(right[right_index])  # put it in the result
            right_index += 1  # advance right index
    merged.extend(left[left_index:])  # append remaining left half
    merged.extend(right[right_index:])  # append remaining right half
    return merged

def merge_sort(arr):  # actual merge sort function

    if len(arr) <= 1:  # if single or zero items => already sorted
        yield arr
    else:  # non-trivial case - we have more items
        mid = len(arr) // 2  # compute midpoint
        left_half = arr[:mid]  # split the array
        right_half = arr[mid:]
        # recursively sort the halves
        left_steps = list(merge_sort(left_half))
        right_steps = list(merge_sort(right_half))

        yield from left_steps
        yield from right_steps
        # merge the sorted halves
        merged = merge(left_steps[-1], right_steps[-1])
        yield merged
```

For merge sort, the visualization is, perhaps, confusing, since it is showing the "small" chunks being sorted. In this case, it is perhaps better to look at the printed steps. See that the first and second items (step 0 and 1) are merged together into one, two-element array at step 2. Then, there is an odd-numbered array (3 items) being split into one item (step 3) and

two items that are split further (steps 4 and 5). The two items are merged into one (step 6). Afterwards, the single element and the two-element arrays are merged into one (step 7). Finally (for this chunk of the whole array), the arrays from steps 2 and 7 are merged into one (step 8).

```
states = test_sort(merge_sort, show_full=True)
```

```
Unsorted array:
[ 6, 19, 14, 10,  7,  6, 18, 10, 10,  3]
 0: [ 6]
 1: [19]
 2: [ 6, 19]
 3: [14]
 4: [10]
 5: [ 7]
 6: [ 7, 10]
 7: [ 7, 10, 14]
 8: [ 6,  7, 10, 14, 19]
 9: [ 6]
10: [18]
11: [ 6, 18]
12: [10]
13: [10]
14: [ 3]
15: [ 3, 10]
16: [ 3, 10, 10]
17: [ 3,  6, 10, 10, 18]
18: [ 3,  6,  6,  7, 10, 10, 10, 14, 18, 19]
```

Quick Sort

Quick sort also divides the array into sub-arrays and recursively sorts these. However, the merge step is essentially reversed, comparing to the merge sort. While merge sort first sorts each half and then sorts and merges them, item-by-item, quick sort first sorts the items into two halves - left half contains all items smaller than any item in the right half. Then, each half is sorted individually. Finally, the sorted halves can be directly concatenated to form a fully sorted array.

The sorting to left (low) and right (high) halves/chunks, a pivot is first chosen. This is an item from the array that will be used to "judge" the rest of the values - values smaller than pivot will go to the left chunk, while values larger than pivot will go to the right chunk.

In a simple implementation, the pivot is chosen as the midpoint element of the array. It's probably obvious that if this item has "unfavorable" value, it can cause issues. If the pivot is incidentally the smallest element in the array, the left chunk will have zero items, while the right one will contain the whole remaining array. In this case, the current sorting step is "wasted", producing almost no change to the array (basically, doing insert sort with the pivot as the key).

Because of this, the worst case (e.g, always choosing the lowest element as pivot) time complexity of quick sort is $O(n^2)$. Nonetheless, if the pivot is chosen well (e.g., using some heuristic or a priori information about the data), the best case time complexity is $O(n \log n)$. In practice it can be faster than merge sort, since the merging process is more efficient. It is also possible to make an in-place sorting implementation of quick sort, when necessary.

```
def quick_sort(arr):
    if len(arr) <= 1:  # trivial case 0 or 1 items => sorted
        yield arr  # just return it
    else:
        pivot = arr[len(arr) // 2]  # choose pivot
        left = [x for x in arr if x < pivot]  # make lower-than pivot chunk
        middle = [x for x in arr if x == pivot]  # make equal-to pivot chunk
        right = [x for x in arr if x > pivot]  # make greater-than pivot chunk

        yield arr

        left_steps = list(quick_sort(left))  # sort the chunks
        right_steps = list(quick_sort(right))

        yield from left_steps
        yield middle
        yield from right_steps
        # merge the sorted chunks (and the middle chunk)
        merged = left_steps[-1] + middle + right_steps[-1]
        yield merged
```

Like for the merge sort, quick sort visualization is also more difficult to follow and it is better to analyze the printed steps. First, the pivot is chosen - in this case as `len(arr) // 2` (index 5 for array of length 10). The arrays is then split into 3 chunks: Chunks 1: all items smaller than pivot (step 1) Chunks 2: all items equal to pivot (step 2) Chunks 3: all items larger than pivot (step 3) These chunks are then also sorted by quick sort and merged.

```
states = test_sort(quick_sort)
```

```
Unsorted array:
```

```
 0: [ 6, 19, 14, 10,  7,  6, 18, 10, 10,  3]
 1: [ 3]
 2: [ 6,  6]
 3: [19, 14, 10,  7, 18, 10, 10]
 4: []
 5: [ 7]
 6: [19, 14, 10, 18, 10, 10]
 7: [14, 10, 10, 10]
 8: []
 9: [10, 10, 10]
10: [14]
11: [10, 10, 10, 14]
12: [18]
13: [19]
14: [10, 10, 10, 14, 18, 19]
15: [ 7, 10, 10, 10, 14, 18, 19]
16: [ 3,  6,  6,  7, 10, 10, 10, 14, 18, 19]
```

Heap Sort

Heap sort uses a **heap** representation of the data - essentially a binary tree in "linear form". Each item at index i is a parent of item at index $2i + 1$ (left child) and $2i + 2$ (right child), and the root of the tree is the item at index 0.

At each step, the heap sort first "heapifies" the array - it checks whether each parent node is larger than the children. If not, it swaps them (so that in the end, parents are always larger than their children).

After the array is made into a valid heap, the root element - element at the index 0 must be the largest item in the array. It is then removed from the array and the remaining array is heapified again. This is repeated until the heap is empty. In practice, the item is not actually removed but placed at the end of the current array and the heap "pointer" is reduced to exclude the sorted part of the array.

Like merge sort, heap sort has also $O(n \log n)$ time complexity in all cases. It has slightly less memory overhead than the merge sort. However, it is "unstable" (e.g., the order of equal items is not guaranteed to be preserved) and the performance does not improve when the array is almost sorted. Actually, for smaller and nearly sorted arrays, insertion sort might be faster.

```python
def heap_sort(a):
    n = len(a)
    states = []

    def make_heap(a, n, i):  # ensure the array `a` is a valid heap
```

```
        largest = i  # go through each parent
        l = 2 * i + 1  # and their children
        r = 2 * i + 2
        if l < n and a[l] > a[largest]:  # and ensure both children are smaller
            largest = l  # otherwise, swap them
        if r < n and a[r] > a[largest]:
            largest = r
        if largest != i:
            a[i], a[largest] = a[largest], a[i]
            states.append(a.copy())  # just for visualization
            make_heap(a, n, largest)  # recursive call for children

    for i in range(n//2 - 1, -1, -1):
        make_heap(a, n, i)  # build max heap
        yield a.copy()

    for i in range(n - 1, 0, -1):
        a[0], a[i] = a[i], a[0]  # yoink the root and put it at the end
        states.append(a.copy())
        yield a.copy()
        make_heap(a, i, 0)
        yield a.copy()
    yield a.copy()
    return a
```

In the visualization, notice how the largest item (in the unsorted part of the array) is first placed at the beginning of the array (i.e., the tree root) and then yanked out and placed at the end of the array. Or, rather, at the end of the heap / start of the sorted end of the array.

```
states = test_sort(heap_sort)
```

```
Unsorted array:
 0: [ 6, 19, 14, 10,  7,  6, 18, 10, 10,  3]
 1: [ 6, 19, 14, 10,  7,  6, 18, 10, 10,  3]
 2: [ 6, 19, 18, 10,  7,  6, 14, 10, 10,  3]
 3: [ 6, 19, 18, 10,  7,  6, 14, 10, 10,  3]
 4: [19, 10, 18, 10,  7,  6, 14,  6, 10,  3]
 5: [ 3, 10, 18, 10,  7,  6, 14,  6, 10, 19]
 6: [18, 10, 14, 10,  7,  6,  3,  6, 10, 19]
 7: [10, 10, 14, 10,  7,  6,  3,  6, 18, 19]
 8: [14, 10, 10, 10,  7,  6,  3,  6, 18, 19]
```

```
 9: [ 6, 10, 10, 10,  7,  6,  3, 14, 18, 19]
10: [10, 10, 10,  6,  7,  6,  3, 14, 18, 19]
11: [ 3, 10, 10,  6,  7,  6, 10, 14, 18, 19]
12: [10,  7, 10,  6,  3,  6, 10, 14, 18, 19]
13: [ 6,  7, 10,  6,  3, 10, 10, 14, 18, 19]
14: [10,  7,  6,  6,  3, 10, 10, 14, 18, 19]
15: [ 3,  7,  6,  6, 10, 10, 10, 14, 18, 19]
16: [ 7,  6,  6,  3, 10, 10, 10, 14, 18, 19]
17: [ 3,  6,  6,  7, 10, 10, 10, 14, 18, 19]
18: [ 6,  3,  6,  7, 10, 10, 10, 14, 18, 19]
19: [ 6,  3,  6,  7, 10, 10, 10, 14, 18, 19]
20: [ 6,  3,  6,  7, 10, 10, 10, 14, 18, 19]
21: [ 3,  6,  6,  7, 10, 10, 10, 14, 18, 19]
22: [ 3,  6,  6,  7, 10, 10, 10, 14, 18, 19]
23: [ 3,  6,  6,  7, 10, 10, 10, 14, 18, 19]
```

Radix Sort

Last sorting algorithm works in a slightly different way than the previous sorts - it's not actually comparing the elements (see that there is not `arr[index] < arr[other_index]` or any other element value comparison). Instead, it sorts by **digits** (can be adapted to sorting strings).

The idea is that the array is sorted by the digits of the number - e.g., 123 is sorted as 1, 2, 3. First, the **most significant digit** is sorted. Depending on the application, it could be the left-most or right-most digit. Here, it is the right-most digit (for 123 it would be 3). Then, the second most significant digit is sorted (for 123 it would be 2), then the third, and so on. For each digit, a list of **buckets** is created and each element is placed in the appropriate bucket. The buckets are then merged to form the final sorted array.

It can be adapted to different numbering systems (binary, hex, …) or even strings. The `RADIX` determines the number of digits or elements per-digit "place" (e.g., 10 for decimal, 2 for binary, 16 for hex).

```python
def radix_sort(arr, RADIX=10):
    placement = 1  # which digit are we processing now
    max_digit = max(arr)  # find the largest number

    while placement < max_digit:  # while there are digits to process
        buckets = [list() for _ in range(RADIX)]  # create RADIX buckets
        for i in arr:  # for each element
            # get the digit at the current placement
            tmp = int((i / placement) % RADIX)  # it can be though of as "hashing"
```

```
            buckets[tmp].append(i)  # add item to the appropriate bucket
        a = 0
        for bucket in buckets:  # bucket-by-bucket
            for item in bucket:  # "empty" the buckets into the array
                arr[a] = item
                a += 1
        # this creates an array sorted by the current digit
        yield arr.copy()
        # advance to the next digit
        placement *= RADIX
    yield arr.copy()
```

Notice that the radix sort requires only two steps, if the elements have at most 2 digits. However, each step is quite laborious and hence it does not mean that radix sort is always faster. The actual time complexity is $O(nk)$ where n is the number of elements and `k` is the number of digits. Therefore, if k is large (relative to n), radix sort might approach quadratic time complexity. Also, of course, radix sort requires for the items to have *digits* or anything equivalent. Otherwise, it can't be used. Although, it can be adapted for fixed-precision floating point numbers, if they are treated as integers (e.g., 3.14 is treated as 314).

```
states = test_sort(radix_sort)
```

```
Unsorted array:
 0: [10, 10, 10,  3, 14,  6,  6,  7, 18, 19]
 1: [ 3,  6,  6,  7, 10, 10, 10, 14, 18, 19]
 2: [ 3,  6,  6,  7, 10, 10, 10, 14, 18, 19]
```

Comparison of sorting algorithms

```
import time
try:
    from tabulate import tabulate
except ImportError:
    tabulate = None

sorting_algorithms = {
    "Bubble Sort": bubble_sort,
    "Insertion Sort": insertion_sort,
    "Selection Sort": selection_sort,
```

```
    "Merge Sort": merge_sort,
    "Quick Sort": quick_sort,
    "Heap Sort": heap_sort,
    "Radix Sort": radix_sort,
}

# Compare for different array lengths and max values
array_lengths = [100, 1000]
max_vals = [100, 100000]

# Compare the run times of different sorting algorithms
for length in array_lengths:
    for max_val in max_vals:
        print(f"Array Length: {length}, Max Value: {max_val}")
        results = []
        arr = np.random.randint(0, max_val, length)
        for name, func in sorting_algorithms.items():
            arr_copy = arr.copy()
            start_time = time.time()
            list(func(arr_copy))
            end_time = time.time()
            run_time = end_time - start_time
            results.append([name, f"{run_time:.6f} seconds"])
        if tabulate is None:
            print('\n'.join([f"{k}: {v}" for (k, v) in results]))
        else:
            print(tabulate(
                results,
                headers=["Sorting Algorithm", "Run Time"], tablefmt="grid")
            )
        print()
```

```
Array Length: 100, Max Value: 100
+---------------------+------------------+
| Sorting Algorithm   | Run Time         |
+=====================+==================+
| Bubble Sort         | 0.005241 seconds |
+---------------------+------------------+
| Insertion Sort      | 0.002116 seconds |
+---------------------+------------------+
| Selection Sort      | 0.001202 seconds |
+---------------------+------------------+
```

```
| Merge Sort           | 0.000450 seconds |
+----------------------+------------------+
| Quick Sort           | 0.000260 seconds |
+----------------------+------------------+
| Heap Sort            | 0.000941 seconds |
+----------------------+------------------+
| Radix Sort           | 0.000172 seconds |
+----------------------+------------------+

Array Length: 100, Max Value: 100000
+----------------------+------------------+
| Sorting Algorithm    | Run Time         |
+======================+==================+
| Bubble Sort          | 0.004056 seconds |
+----------------------+------------------+
| Insertion Sort       | 0.002565 seconds |
+----------------------+------------------+
| Selection Sort       | 0.001197 seconds |
+----------------------+------------------+
| Merge Sort           | 0.000647 seconds |
+----------------------+------------------+
| Quick Sort           | 0.000297 seconds |
+----------------------+------------------+
| Heap Sort            | 0.000952 seconds |
+----------------------+------------------+
| Radix Sort           | 0.000278 seconds |
+----------------------+------------------+

Array Length: 1000, Max Value: 100
+----------------------+------------------+
| Sorting Algorithm    | Run Time         |
+======================+==================+
| Bubble Sort          | 1.746768 seconds |
+----------------------+------------------+
| Insertion Sort       | 0.451123 seconds |
+----------------------+------------------+
| Selection Sort       | 0.155976 seconds |
+----------------------+------------------+
| Merge Sort           | 0.005072 seconds |
+----------------------+------------------+
| Quick Sort           | 0.001399 seconds |
+----------------------+------------------+
| Heap Sort            | 0.027272 seconds |
```

```
+---------------------+------------------+
| Radix Sort          | 0.001246 seconds |
+---------------------+------------------+

Array Length: 1000, Max Value: 100000
+---------------------+------------------+
| Sorting Algorithm   | Run Time         |
+=====================+==================+
| Bubble Sort         | 0.974790 seconds |
+---------------------+------------------+
| Insertion Sort      | 0.438587 seconds |
+---------------------+------------------+
| Selection Sort      | 0.128462 seconds |
+---------------------+------------------+
| Merge Sort          | 0.071643 seconds |
+---------------------+------------------+
| Quick Sort          | 0.004499 seconds |
+---------------------+------------------+
| Heap Sort           | 0.025438 seconds |
+---------------------+------------------+
| Radix Sort          | 0.002785 seconds |
+---------------------+------------------+
```

Sliding window operations

Sliding window operations are a fundamental concept in signal processing and image analysis. They involve applying a function to a subset of data, moving position (start/end indices) of the subset over the entire data set, and computing the output at each position.

Fixed-size window

In the simplest case, a **window** of a fixed size is used.

Moving average

The moving average is a simple operation that replaces each element in an array with the average of its neighboring elements within a fixed-size window. That is, for element at index i and window size w, the moving average is the average of the elements at indices $(i - w//2)$ to $(i+w//2)$. For simplicity, we will consider odd windows sizes. One issue that needs to be solved is what happens if the window "overflows" the array bounds (either from the beginning or the

end). This issue can be done with various padding techniques (zero-padding, DC-padding, circular padding - depends on the desired behavior) or simply limiting the window size.

Here is a Python implementation of the moving average:

```
def moving_average(arr, window_size):
    numel = len(arr)
    half_window = window_size // 2
    avg = np.zeros(numel)
    for i in range(numel):
        start = max(0, i - half_window)
        end = min(numel, i + half_window + 1)
        avg[i] = sum(arr[start:end]) / (end - start)
    return avg

arr = np.random.randint(0, 10, 10)
window_size = 3
print("Original array:", arr)
print("Moving average:", moving_average(arr, window_size))
```

```
Original array: [5 1 0 6 6 9 9 1 1 0]
Moving average: [3.         2.         2.33333333 4.         7.         8.
 6.33333333 3.66666667 0.66666667 0.5       ]
```

Alternatively, we can pad the array and compute the average only on "valid" elements. This has the advantage that it requires slightly less computations per step but also, provides a different result at the "edges" of the array.

```
def moving_average(input_arr, window_size):
    numel = len(input_arr)
    half_window = window_size // 2
    # zero-pad the array
    arr = np.zeros(numel + 2 * half_window)
    arr[half_window:-half_window] = input_arr
    avg = np.zeros(numel)
    for i in range(numel):
        avg[i] = sum(arr[i:i + window_size]) / window_size
    return avg

arr = np.random.randint(0, 10, 10)
window_size = 5
print("Original array:", arr)
print("Moving average:", moving_average(arr, window_size))
```

```
Original array: [9 7 7 0 7 8 3 0 0 2]
Moving average: [4.6 4.6 6.  5.8 5.  3.6 3.6 2.6 1.  0.4]
```

While this implementation is straight forward, it is not ideal, especially for larger window sizes. The reason is that for each position in the array, all the "surrounding" elements need to be summed. Can we do better?

```
def rolling_average(input_arr, window_size):
    numel = len(input_arr)
    half_window = window_size // 2
    # zero-pad the array
    arr = np.zeros(numel + 2 * half_window)
    arr[half_window:-half_window] = input_arr
    avg = np.zeros(numel)
    # compute the first sum and average
    wsum = sum(arr[:window_size])
    avg[0] = wsum / window_size
    for i in range(1, numel):
        wsum = wsum - arr[i - 1] + arr[i + window_size - 1]
        avg[i] = wsum / window_size
    return avg

print("Original array:", arr)
print("Moving average:", rolling_average(arr, window_size))
```

```
Original array: [9 7 7 0 7 8 3 0 0 2]
Moving average: [4.6 4.6 6.  5.8 5.  3.6 3.6 2.6 1.  0.4]
```

In this version, instead of computing the sum of elements from the current window from scratch, we compute "a rolling sum". We keep the sum from the last step. To compute the sum for the current window, we *remove*, i.e. subtract the left-most ("oldest") element and add in the right-most element (the new one that the window has "moved over" to). This way, we only need two addition operations per step (plus division, when computing the average).

Speed comparison:

```
large_array = np.random.randint(0, 10, 1000)
window_size = 15
print("Run time for 'simple' moving average:")
%timeit moving_average(large_array, window_size)
print("Run time for improved moving average:")
%timeit rolling_average(large_array, window_size)
```

```
print(
    "Are the results equal:",
    np.allclose(
        moving_average(large_array, window_size),
        rolling_average(large_array, window_size)
))
```

```
Run time for 'simple' moving average:
2.98 ms ± 112  s per loop (mean ± std. dev. of 7 runs, 100 loops each)
Run time for improved moving average:
668  s ± 162  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
Are the results equal: True
```

Variable-size window

In some cases, it is useful to traverse an array with a variable-size window. For example, finding maximum sub-array, constrained by some max-sum value. That is, the we are looking for a sub-array `sarr = arr[start:end]` for which `sum(sarr) <= max_sum`.

For example, in the given array:

```
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The maximum sub-array for `max_sum = 20` is `[2, 3, 4, 5, 6]`, summing to (exactly) 20.

We can, of course, **brute force** it by checking all possible combinations:

```
def find_max_subarray_bruteforce(arr, max_sum):
    numel = len(arr)
    best_sum = 0
    best_sub = []
    for start in range(numel):
        for end in range(start + 1, numel + 1):
            current_sum = sum(arr[start:end])
            if current_sum <= max_sum and current_sum > best_sum:
                best_sum = current_sum
                best_sub = arr[start:end]
    if len(best_sub) == 0:
        return None
    return best_sub, best_sum

print("Maximum sub-array:", find_max_subarray_bruteforce(arr, 20))
```

```
Maximum sub-array: ([2, 3, 4, 5, 6], 20)
```

But this is not the most efficient way to do it. Instead, we can use a sliding window with variable size, bounded by `start` and `end` positions with the following rules:

1. The window size is initially set to 1.
2. If the sum of the current window is less than `max_sum`, increase `end` by 1. This adds another (new) element to the sum, increasing it.
3. If the sum of the current window is greater than or equal to `max_sum`, increase `start` by 1. This removes an element from the sum, decreasing it. Technically, if it is equal to `max_sum`, we could end, if we are looking for only one solution.

```python
def find_max_subarray(arr, max_sum):
    numel = len(arr)
    start = 0
    end = 1
    best_sum = 0
    best_start, best_end = 0, 0
    while end <= numel:
        current_sum = sum(arr[start:end])  # we could also use the rolling sum
        if current_sum <= max_sum and current_sum > best_sum:
            best_sum = current_sum
            best_start, best_end = start, end
        if current_sum < max_sum:
            end += 1
        else:
            start += 1
    best_sub = arr[best_start:best_end]
    if len(best_sub) == 0:
        return None
    return best_sub, best_sum

print("Maximum sub-array:", find_max_subarray(arr, 20))
```

```
Maximum sub-array: ([2, 3, 4, 5, 6], 20)
```

Speed comparison:

```python
N = 100
large_array = np.random.randint(0, 10, N)
max_sum = N >> 2
```

```
print("Run time for brute force max subarray:")
%timeit find_max_subarray_bruteforce(large_array, max_sum)
print("Run time for sliding window max subarray:")
%timeit find_max_subarray(large_array, max_sum)
a = find_max_subarray_bruteforce(large_array, max_sum)
b = find_max_subarray(large_array, max_sum)
if a is not None and b is not None:
    print("Are the results equal:", np.allclose(a[0], b[0]))
else:
    print("Results are None. (couldn't find a valid subarray)")
```

```
Run time for brute force max subarray:
22.4 ms ± 1.43 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
Run time for sliding window max subarray:
431  s ± 67.1  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
Are the results equal: True
```

Convolution

Convolution is a more general operation that combines two arrays by sliding one array over the other and computing the dot product at each position. It is often used in image processing and signal processing. Many different tasks can be solved using convolution - peak/edge detection, filtering, "discrete derivation", etc.

Similarly to sliding window, one has to deal with the array boundary issue - either compute the indices so that the don't overflow the "edge" of the array or pad the array.

We will implement a helper function to zero-pad and DC-pad (repeat the boundary values) an array:

```
def zero_pad_1d(arr, kernel_size):
    pad_size = kernel_size // 2
    # use NumPy pad function
    return np.pad(arr, pad_size, mode='constant', constant_values=0)

def dc_pad_1d(arr, kernel_size):
    pad_size = kernel_size // 2
    return np.pad(arr, pad_size, mode='edge')
```

Here, we don't pad "manually" but make use of the `NumPy.pad` function.

1D convolution

Here is an implementation of 1D convolution:

```
def convolve1d(in_arr, kernel, pad_type='zero'):
    kernel_size = len(kernel)
    numel = len(in_arr)
    if pad_type == 'zero':  # first, resolve the padding
        arr = zero_pad_1d(in_arr, kernel_size)
    elif pad_type == 'dc':
        arr = dc_pad_1d(in_arr, kernel_size)
    result = np.zeros(numel)

    for i in range(numel):  # convolve
        result[i] = sum(arr[i:i + kernel_size] * kernel)
    return result
```

The implementation for 1D case is quite simple. For each slice of the input array, multiply it with the kernel and sum the result - essentially a dot product.

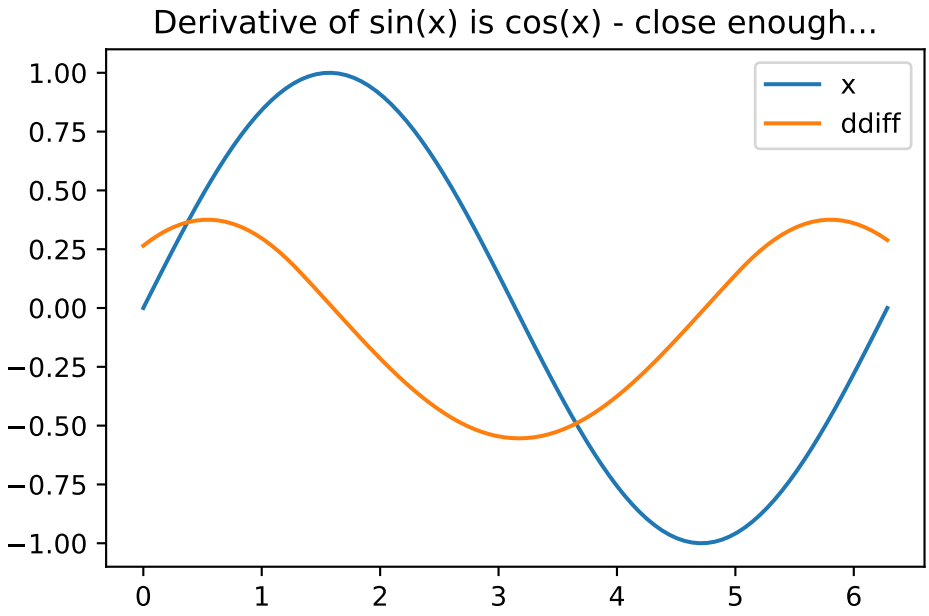
Let's see some practical use-cases of 1D convolution.

Discrete "Derivative"

A discrete approximate derivative can be computed by a kernel that essentially computes the difference of the "function", represented by the array values, at the current index. Such kernels are, for example `[-1, 1]` or `[-1, 0, 1]` - add the next value, subtract the previous. To achieve a smoother result, larger kernel can be used (computes difference over a larger window). For a kernel size independent result, the kernel needs to be normalized. Keep in mind that this is an approximation and is typically used to "detect" when there is a positive or negative (trend) change in the signal. Not the actual derivative.

```
x_vals = np.linspace(0, np.pi * 2, 100)
arr = np.sin(x_vals)
diff_kernel_size = 20
kernel = np.r_[[-1] * diff_kernel_size, [1] * diff_kernel_size]  # "derivation" kernel
kernel = kernel / len(kernel)
ddiff = convolve1d(arr, kernel, "dc")

import matplotlib.pyplot as plt
plt.plot(x_vals, arr, label="x")
plt.plot(x_vals, ddiff, label="ddiff")
```

```
plt.title("Derivative of sin(x) is cos(x) - close enough...")
plt.legend()
plt.show()
```



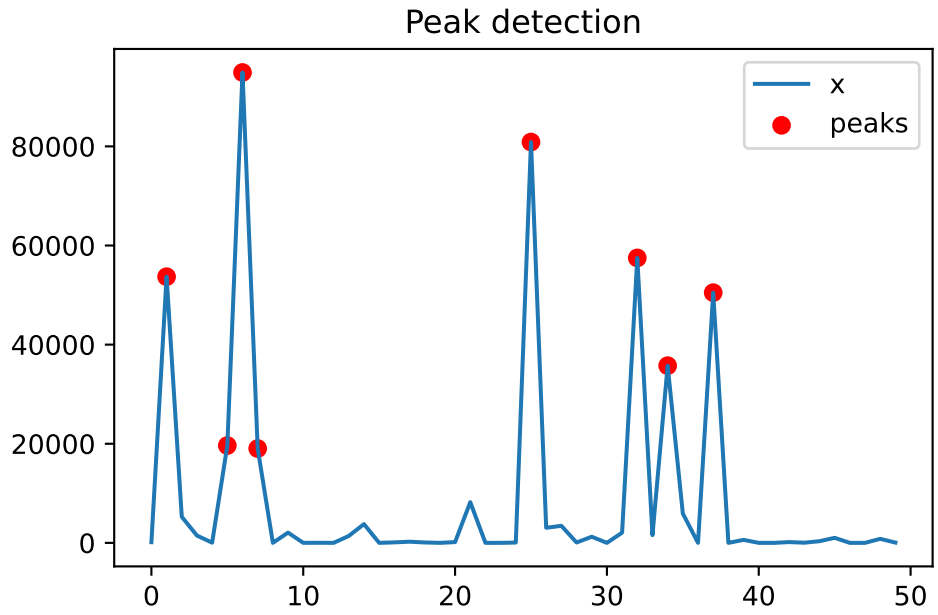
Peak detection

Sometimes, peaks or otherwise prominent values (local maxima) need to be detected. A kernel to detect peaks subtracts the surrounding values from the *central* value. The central value is also multiplied so that if it is larger than the surrounding values, the result will be positive.

```
arr = np.exp(np.random.rand(50)) ** 12
peak_window = 15
# peak kernel subtracts the surrounding of a value from the value itself
kernel = np.r_[[-1] * peak_window, peak_window * 2 + 1, [-1] * peak_window]
peak_response = convolve1d(arr, kernel, "zero")
peaks = np.where(peak_response > 0)[0]

print("Peaks:", peaks)
x = np.arange(len(arr))
plt.plot(x, arr, label="x")
plt.scatter(peaks, arr[peaks], label="peaks", color="red")
plt.title("Peak detection")
```

```
plt.legend()
plt.show()
```

```
Peaks: [ 1  5  6  7 25 32 34 37]
```



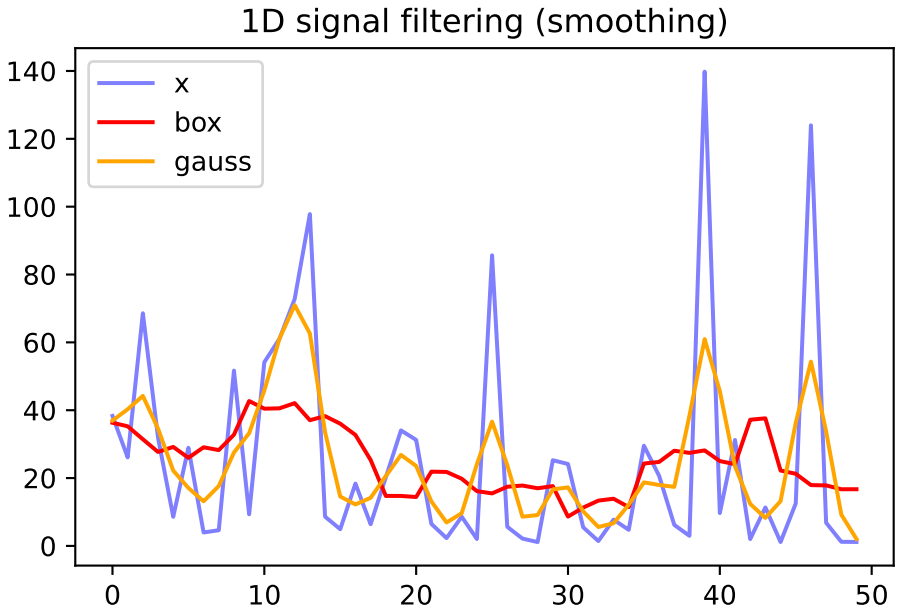
The result is not always perfect and typically requires some additional processing. Nonetheless, it is a fast way to detect at least potential candidates for local maxima.

Filtering (smoothing)

Using the convolution, a signal can be "smoothed". To smooth a signal, box (average) or Gaussian kernels can be used (other smoothing kernels exist). Smoothing kernels are often normalized to unit sum to keep the total "signal-strength" unchanged (otherwise, the kernel would also "amplify" the signal).

```
arr = np.exp(np.random.rand(50)) ** 5
box_size = 9
box_kernel = np.ones(box_size) / box_size
gauss_kernel = np.exp(-0.5 * np.arange(-4, 5) ** 2) / np.sqrt(2 * np.pi)
box_smoothed = convolve1d(arr, box_kernel, "dc")
gauss_smoothed = convolve1d(arr, gauss_kernel, "dc")

plt.plot(arr, label="x", color="b", alpha=0.5)
```

```
plt.plot(box_smoothed, label="box", color="red")
plt.plot(gauss_smoothed, label="gauss", color="orange")
plt.title("1D signal filtering (smoothing)")
plt.legend()
plt.show()
```



2D convolution

2D convolution is a core operation of image processing. It is similar to 1D convolution, but the kernel is applied to a 2D array.

Like before, we need padding function:

```
def pad_zero_2d(image, kernel_height, kernel_width):
    # Calculate the padding sizes
    pad_height = kernel_height // 2
    pad_width = kernel_width // 2

    # Pad the image with zeros
    padded_image = np.pad(
        image,
        ((pad_height, pad_height), (pad_width, pad_width)),
        mode='constant', constant_values=0
```

```
    )
    return padded_image

def pad_dc_2d(image, kernel_height, kernel_width):
    # Calculate the padding sizes
    pad_height = kernel_height // 2
    pad_width = kernel_width // 2

    padded_image = np.pad(image,
        ((pad_height, pad_height), (pad_width, pad_width)),
        mode='edge'
    )
    return padded_image
```

The 2D convolution loops over both dimensions of the input image and applies the kernel to each region of the image. As in the 1D convolution, dot product is used to compute the convolution. Although, in this case, it's a dot product between two "vectorized" (flattened) matrices.

```
def convolve2d(image, kernel, padding_type='zero'):
    # Get the dimensions of the image and kernel
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape

    # Pad the image based on the padding type
    if padding_type == 'zero':
        padded_image = pad_zero_2d(image, kernel_height, kernel_width)
    elif padding_type == 'dc':
        padded_image = pad_dc_2d(image, kernel_height, kernel_width)
    else:
        raise ValueError("Invalid padding_type. Use 'zero' or 'dc'.")

    # Calculate the output dimensions
    output_height = image_height
    output_width = image_width

    # Initialize the output array
    output = np.zeros((output_height, output_width))

    # Perform the convolution
    for i in range(output_height):
        for j in range(output_width):
```

```
            # Extract the region of interest from the padded image
            region = padded_image[i:i+kernel_height, j:j+kernel_width]
            # Perform element-wise multiplication and sum the results
            output[i, j] = np.sum(region * kernel)

    return output
```

Let's test the convolution on a small matrix and a kernel.

```
# Create a simple 5x5 image
image = np.arange(25).reshape(5, 5)

# Create a simple 3x3 kernel
kernel = np.array([
    [1, 0, -1],
    [1, 0, -1],
    [1, 0, -1]
])

# Perform the convolution with zero padding
result_zero = convolve2d(image, kernel, padding_type='zero')
print("Convolved Image with Zero Padding:")
print(result_zero)

# Perform the convolution with dc padding
result_dc = convolve2d(image, kernel, padding_type='dc')
print("Convolved Image with DC Padding:")
print(result_dc)
```

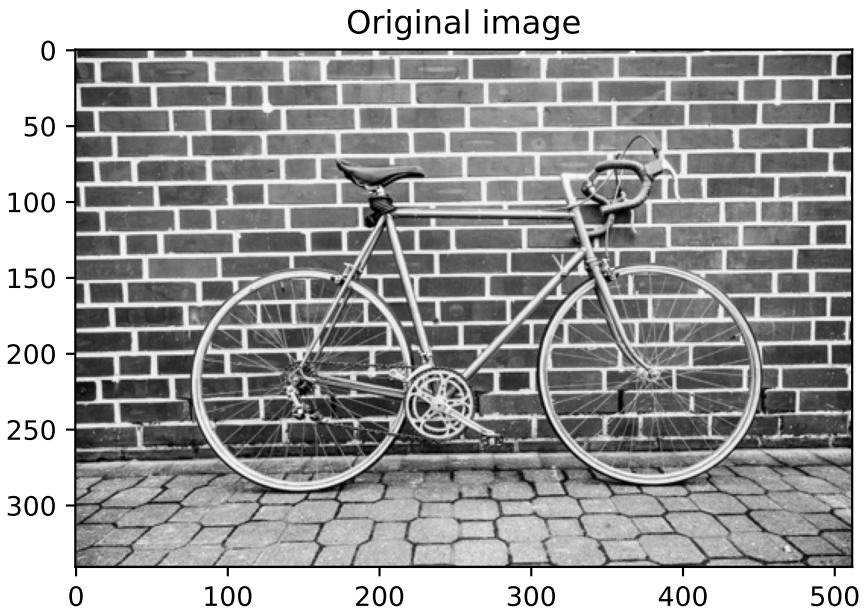
```
Convolved Image with Zero Padding:
[[ -7.  -4.  -4.  -4.  11.]
 [-18.  -6.  -6.  -6.  24.]
 [-33.  -6.  -6.  -6.  39.]
 [-48.  -6.  -6.  -6.  54.]
 [-37.  -4.  -4.  -4.  41.]]
Convolved Image with DC Padding:
[[-3. -6. -6. -6. -3.]
 [-3. -6. -6. -6. -3.]
 [-3. -6. -6. -6. -3.]
 [-3. -6. -6. -6. -3.]
 [-3. -6. -6. -6. -3.]]
```

To use 2D convolution on image, we need to be able to load some images. For this we will use the `PIL` library.

```
from PIL import Image
import numpy as np
import os

def load_png_image(file_path, fixed_width=512):
    # Open the image file
    image = Image.open(file_path)
    image = image.convert("L")

    original_width, original_height = image.size
    # Calculate the new height while maintaining the aspect ratio
    new_height = int((fixed_width / original_width) * original_height)
    # Resize the image
    image = image.resize((fixed_width, new_height))
    # Convert the image to a NumPy array
    image_array = np.array(image)
    return image_array
```

Now we can load an image and display it. The loading function also converts the image to gray-scale, since otherwise, we would have to deal with color channels. Additionally, it resizes the image, so that the convolution does not take to much time if the image is too large.

```
# Load the image
# image_file = os.path.join(os.path.dirname(__file__), 'bike.png')
image_file = 'bike.png'
image = load_png_image(image_file)

# Display the image
plt.imshow(image, cmap='gray')
plt.title("Original image")
plt.show()
```



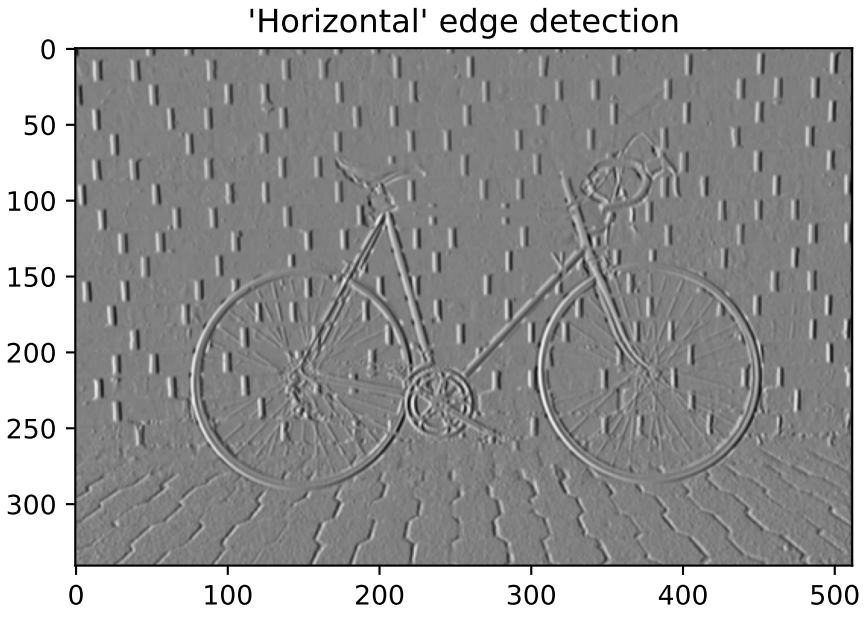
Now, we are ready to test 2D convolution on the image.

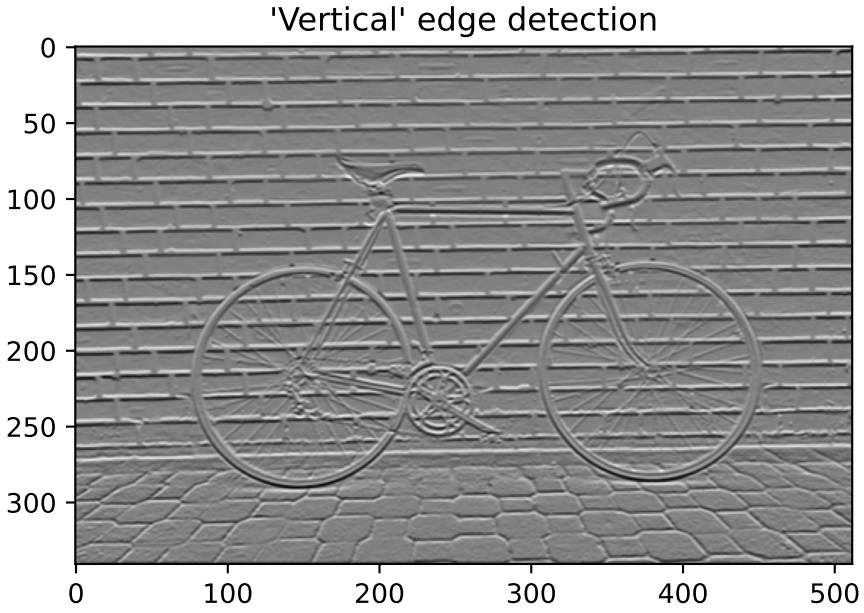
Sobel edge detection

There are several different ways to compute edge detection on images. The specific algorithm/kernel depend on what is required. In this example, we will use the Sobel edge detection algorithm. Sobel kernels are directional - they detect horizontal or vertical edges - or rather horizontal or vertical **gradients** in the image, which corresponds with the direction perpendicular to the edge.

```
# Define the Sobel kernels
sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
sobel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

# Apply the Sobel kernels to the image
sobel_x_result = convolve2d(image, sobel_x, padding_type='dc')
sobel_y_result = convolve2d(image, sobel_y, padding_type='dc')

plt.imshow(sobel_x_result, cmap='gray')
plt.title("'Horizontal' edge detection")
plt.show()

plt.imshow(sobel_y_result, cmap='gray')
```

```
plt.title("'Vertical' edge detection")
plt.show()
```

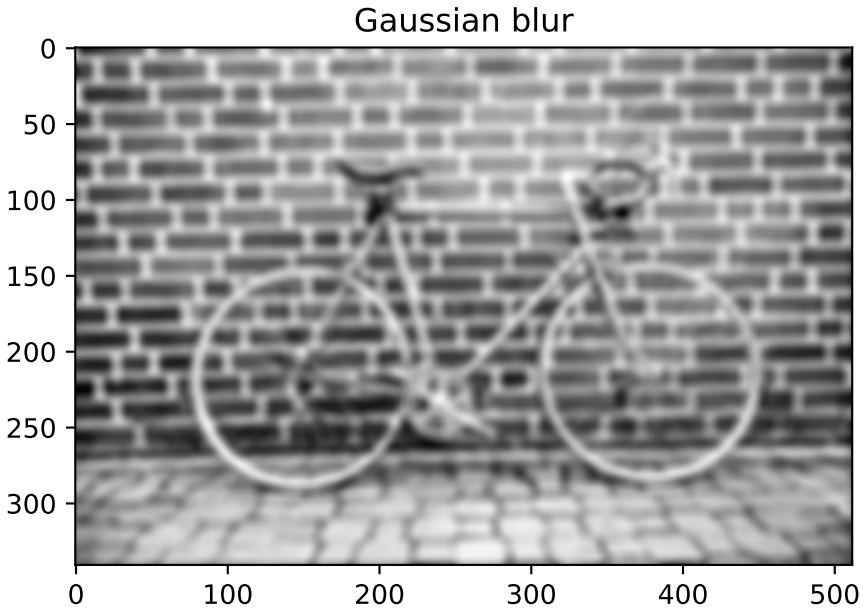




Gaussian blur

Much like for 1D signal smoothing, images can also be smoothed. Let's see an example using a Gaussian kernel.

```
# Define the Gaussian kernel
kernel_size = 11  # kernel size in both dimensions
sigma = 3  # sigma for the Gaussian
kernel = np.zeros((kernel_size, kernel_size))
# fill in the values for the Gaussian kernel
for i in range(kernel_size):
    for j in range(kernel_size):
        x = i - kernel_size // 2
        y = j - kernel_size // 2
        kernel[i, j] = (  # eq. for Gaussian
            1 / (2 * np.pi * sigma**2)) * np.exp(-((x**2 + y**2) / (2 * sigma**2))
        )

# apply the Gaussian blur to the image
gaussian_result = convolve2d(image, kernel, padding_type='dc')

plt.imshow(gaussian_result, cmap='gray')
plt.title("Gaussian blur")
plt.show()
```



Sparse matrices

In some cases, there might be a need to represent and work with so-called **sparse** data. For example, edges in a weighted graph might be represented by a matrix, where a row i and column j represents the weight of the edge from node i to node j (e.g., current flowing from node i to node j in an electrical circuit, or the distance between node i and node j). Another example would be a binary image (e.g., a typical result from edge or other types of detection). In both cases, the underlying data can be represented by a matrix. However, typically, the number of elements "of interest", i.e., elements with non-zero values, is much smaller than the total number of elements. That means, the matrix is **sparse** and, using "classical" 2D array, will consume a lot of "needless" memory. For example, a 9×9 matrix with the following values:

Table 1: Sparse matrix example

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	0	5	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	4	0	0	0	0	0
3	0	2	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	6	0	7	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	12

In Python, we can store the data in a 2D array:

```
import numpy as np

# it's mostly zeros, let's initialize with zeros
dense_matrix = np.zeros((9, 9))
# and only "define" non-zero values
dense_matrix[0, 1] = 5
dense_matrix[2, 3] = 4
dense_matrix[3, 1] = 2
dense_matrix[5, 3] = 6
dense_matrix[5, 5] = 7
dense_matrix[7, 6] = 1
dense_matrix[8, 8] = 12
```

There are a total of 9 * 9 = 81 elements in the matrix. However, only 7 of them have non-zero values. if we consider the case where we would need to use 32-bit integers (4 bytes) to

store the data (numbers between -2^31 and 2^31), the matrix will consume 9 * 9 * 4 = 324 bytes, while storing only 7 * 4 = 28 bytes "worth" of data. Roughly 300 extra bytes is not a big of a deal. However, imagine you have a database of 1 million people representing some kind of asymmetric transactions between them. If on average, each person "deals" with 100 other people and you need *only* 1 byte to store the data about a transaction, then all of the "interesting" data will require:

$$1,000,000\ people \times 100\ transactions \times 1\ byte = 100,000,000\ bytes \approx 100\ MB$$

However, the full matrix will have:

$$1,000,000\ rows \times 1,000,000\ columns \times 1\ byte = 1,000,000,000,000\ bytes \approx 1\ TB$$

That's quite a lot of storage space required to store mostly zeros. To deal with this, sparse-matrix representations are used. There are multiple different approaches, we will show two of them.

Coordinate list (COO)

Perhaps the simplest way to store matrix data with mostly empty elements, is to store the indices of the non-zero elements, along with the corresponding values. E.g., considering the example from the table above, we would store the following:

Row	Column	Value
0	1	5
2	3	4
3	1	2
5	3	6
5	5	7
7	6	1
8	8	12

Representing the sparse matrix in this way is called "coordinate list" (COO).

Let's make a function that would convert a dense matrix to COO:

```
def dense_to_coo(matrix, empty_value=0):
    data = []  # here we will store our data
    row = []  # here we will store the row indices
    col = []  # here we will store the column indices
```

```
    for i, d_row in enumerate(matrix):  # iter over the rows
        for j, d_item in enumerate(d_row):  # iter over the items in a row
            if d_item != empty_value:
                data.append(d_item)
                row.append(i)
                col.append(j)
    data = np.array(data, dtype=matrix.dtype)  # convert to match the input type
    return data, row, col
```

Notice that the function also allow us to define what is considered as "non-zero" (or empty) value. In some cases, the "base" value need not be zero.

Now, demonstrate the function on our example matrix:

```
A_data, A_row, A_col = dense_to_coo(dense_matrix)
print(A_data)
print(A_row)
print(A_col)
```

```
[ 5.  4.  2.  6.  7.  1. 12.]
[0, 2, 3, 5, 5, 7, 8]
[1, 3, 1, 3, 5, 6, 8]
```

Now, let's create a simple Python class to store COO data and "attach" the function to it:

```
class COO:
    def __init__(self, dense_data, empty_value=0):
        # not the nicest way of doing thins but it'll do
        self.data, self.row, self.col = dense_to_coo(dense_data, empty_value)
        # let's remember the shape of the original matrix
        self.shape = dense_data.shape

    def __repr__(self):
        # just to have a nice print output
        return f"COO(\n\tdata={self.data}\n\trow={self.row}\n\tcol={self.col}\n)"

    @classmethod
    def from_coo_arrays(cls, data, row, col, shape):
        # create an empty COO object
        c = cls(np.zeros(shape))
        # attach the data
```

```
        c.data = np.array(data)
        c.row = row
        c.col = col
        return c

    def to_dense(self):
        # prepare the dense matrix
        dense = np.zeros(shape=self.shape, dtype=self.data.dtype)
        # reconstruct the dense matrix
        for d, r, c in zip(self.data, self.row, self.col):
            dense[r, c] = d
        return dense
```

Let's test that the code works properly:

```
coo = COO(dense_matrix)
print(coo)
print("COO to dense:\n", coo.to_dense())
print("Dense matrix:\n", dense_matrix)
```

```
COO(
    data=[ 5.  4.  2.  6.  7.  1. 12.]
    row=[0, 2, 3, 5, 5, 7, 8]
    col=[1, 3, 1, 3, 5, 6, 8]
)
COO to dense:
 [[ 0.  5.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  4.  0.  0.  0.  0.  0.]
 [ 0.  2.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  6.  0.  7.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 12.]]
Dense matrix:
 [[ 0.  5.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  4.  0.  0.  0.  0.  0.]
 [ 0.  2.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

```
 [ 0.  0.  0.  6.  0.  7.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 12.]]
```

Obviously, in this representation, we need some extra memory to store the coordinates. The (required) data type of the coordinates depends on the size of the matrix. In this example, a byte is enough. Therefore, we would need $1B + 1B + 4B = 6B$ for each datapoint. In total, this will require $7 \times 6B = 42B$ of memory. For the large example above with the 1 million people, we have to use at least 32-bit integers. In that case, the total memory requirement would come to:

$$1,000,000\ people \times 100\ transactions \times (2 \times 4B + 1B) \approx 900\ MB$$

Still a decent saving. Of course, since there is some **memory overhead** required, it is not always the best choice. If we, for simplicity, consider both the coordinates and the data to be represented by the same data type, then COO is only **spatially effective** if the number of non-zero elements (NNZ) is at least 3 times smaller than the number of elements, i.e.:

$$NNZ < \frac{N \times N}{3}$$

However, because of the extra **computational overhead**, it's best to reserve the use of sparse matrices for when NNZ is much smaller than that and the overall matrix size is quite large. The specific values will depend on the specific use-case (e.g., what kind of operations are going to be performed on the matrix).

Operations on COO sparse matrices

Some operations on sparse matrices, e.g., scalar multiplication, are quite straight-forward and provide apparent speedup: for an $M \times N$ matrix, only NNZ multiplications need to be performed, instead of $M \times N$.

However, other operations, like matrix multiplications are more complex to implement.

```python
def coo_matmul(sparse_A, sparse_B):
    # Initialize the result data, row, and col arrays
    result_data = []
    result_row = []
    result_col = []
    result_shape = (sparse_A.shape[0], sparse_B.shape[1])
```

```
    # The two outer loops for the result matrix items
    for i in range(result_shape[0]):
        for j in range(result_shape[1]):
            # Initialize the sum for the current element
            sum = 0
            # The two inner loops iterate of the input matrices
            # Iterate over NNZ of sparse_A in the current row
            for k, (d, r, c) in enumerate(
                zip(sparse_A.data, sparse_A.row, sparse_A.col)):
                if r == i:
                    # Iterate over NNZ of sparse_B in the current column
                    for other_k, (other_d, other_r, other_c) in enumerate(
                        zip(sparse_B.data, sparse_B.row, sparse_B.col)):
                        if other_c == j and other_r == c:
                            # Add the product to the sum
                            sum += d * other_d
            # If the sum is non-zero, add it to the result
            if sum != 0:
                result_data.append(sum)
                result_row.append(i)
                result_col.append(j)

    result = COO.from_coo_arrays(result_data, result_row, result_col, result_shape)
    return result
```

To test our methods, we need to create some sparse data. The following function generates a matrix with approximately $NNZ = n\_rows \times n\_cols \times fill\_rate$ non-zero elements.

```
def make_sparse_data(n_rows, n_cols, fill_rate = 0.01):
    # make a matrix with approx. fill_rate * 100 % of non-zero elements
    if fill_rate * n_rows * n_cols < 1:
        fill_rate = 1 / (n_rows * n_cols)  # at least 1 element
    data = np.random.rand(n_rows, n_cols)
    while data.min() > fill_rate:  # at least 1 element above fill_rate
        data = np.random.rand(n_rows, n_cols)
    data[data > fill_rate] = 0
    return data
```

Now, we can test that the code works properly:

```
N = 100
# prepare the data
A = make_sparse_data(N, N, 0.01)
B = make_sparse_data(N, N, 0.01)
# dense matrix multiplication
C = A @ B

A_sparse = COO(A)
B_sparse = COO(B)
# sparse matrix multiplication
C_sparse = coo_matmul(A_sparse, B_sparse)

C_densified = C_sparse.to_dense()

print("Check if the results are equal:", np.allclose(C, C_densified))
```

```
Check if the results are equal: True
```

Of course, runtime-wise, our four loop-based implementation is not going to beat the optimized, multi-threaded NumPy matrix multiplication, that is actually, implemented in C. Still, storage-wise, we are still saving some space.

Speed comparison:

```
%timeit -n 1 -r 1 coo_matmul(A_sparse, B_sparse)
%timeit -n 1 -r 1 A @ B
```

```
602 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
221  s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Size comparison:

```
import sys
print(f"Size of A: {sys.getsizeof(A) / 1e6} MB")
print("Size of A_sparse: "
      f"{(sys.getsizeof(A_sparse.data) + sys.getsizeof(A_sparse.row) + sys.getsizeof(A_sparse
```

```
Size of A: 0.080128 MB
Size of A_sparse: 0.002784 MB
```

Again, the size difference is going to be relevant for much larger matrices.

Now, this is not the most efficient implementation. There are a few "speed-ups" that can be implemented. For example, we could make sure the indices are sorted and only "pick" the current row/column in the inner loops.

Compressed sparse rows (CSR)

One representation of sparse matrices that allows for faster matrix multiplications (and other operations) is compressed sparse rows (CSR).

It is similar to COO in that only the data and indices are stored. For CSR, the data and columns of each data point are stored in in exactly the same way as for COO. However, the row indices are not stored directly. Instead, only **pointers** to where each row starts are stored. For example, the following matrix:

1	0	2
4	5	0
0	0	0
0	6	7

Will be stored as:

```
values = [1, 2, 4, 5, 6, 7]
col_indices = [0, 2, 0, 1, 1, 2]
row_pointers = [0, 2, 4, 4, 6]
```

The `row_pointers` indicate that row `0` starts at index `0` and row `1` starts at index `2`. This means that `values[0:2]` at `col_indices[0:2]` represent the data in the first row (row `0`). For empty rows, the `row_pointers` will be set to the index of the next row. E.g., row `2` in our example has index `4`, the same as row `3`, indicating, it's empty. The last value in row indices indicate the end of the last row. This is per-convention to make it clear whether the last row is empty or not.

The advantage of this representation is that rows can be retrieved very quickly:

```
def print_csr_row(values, col_indices, row_pointers):
    row = 1
    start = row_pointers[row]
    end = row_pointers[row + 1]
    print(f"Row {row}:")
    print("Coumn indices:", col_indices[start:end])
```

```
    print("Row values:", values[start:end])

print_csr_row(values, col_indices, row_pointers)
```

```
Row 1:
Coumn indices: [0, 1]
Row values: [4, 5]
```

This enables, for example, much more efficient matrix-vector multiplication..

CSR in practice

While it's nice to know the "inner workings" of sparse matrices, there are also off-the-shelf implementations available in Python.

For the following example, the `SciPy` library is required (`pip install scipy`).

```
import numpy as np
from scipy.sparse import csr_matrix

# Create two dense matrices
N = 2000
A = make_sparse_data(N, N, 0.01)
B = make_sparse_data(N, N, 0.01)

# Perform matrix multiplication using NumPy
C = np.dot(A, B)

# Create two sparse matrices
A_sparse = csr_matrix(A)
B_sparse = csr_matrix(B)

# Perform matrix multiplication using SciPy
C_sparse = A_sparse.dot(B_sparse)

# Check if the results are equal
print(np.allclose(C, C_sparse.toarray()))
```

```
True
```

```
print("Run time of dense matrix multiplication:")
%timeit -n 1 -r 1 np.dot(A, B)
print("Run time of sparse matrix multiplication:")
%timeit -n 1 -r 1 A_sparse.dot(B_sparse)
```

```
Run time of dense matrix multiplication:
256 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Run time of sparse matrix multiplication:
10.9 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```