Anwendungsbaustein Auswertung von fds-Daten

Lukas Arnold Simone Arnold Florian Bagemihl
Matthias Baitsch Marc Fehr Franca Hollmann
Maik Poetzsch Sebastian Seipel

2025-11-24

Inhaltsverzeichnis

Preamble

Zitiervorschlag

Arnold, Lukas, Simone Arnold, Matthias Baitsch, Marc Fehr, Franca Hollmann, Maik Poetzsch, und Sebastian Seipel. 2025. „Bausteine Computergestützter Datenanalyse. Anwendungsbaustein Auswertung von fds-Daten". https://github.com/bausteine-der-datenanalyse/a-auswertung-fds-daten.

BibTeX-Vorlage

```
@misc{BCD-fds-daten-2025,
 title={Bausteine Computergestützter Datenanalyse. Anwendungsbaustein Auswertung von fds-Date
 author={Arnold, Lukas and Arnold, Simone and Baitsch, Matthias and Fehr, Marc and Hollmann,
 year={2025},
 url={https://github.com/bausteine-der-datenanalyse/a-auswertung-fds-daten}}
```

Intro

Voraussetzungen

- Grundlagen Python
- Einbinden von zusätzlichen Paketen
- Arbeiten mit NumPy
- Arbeiten mit Pandas
- Plotten mit Matplotlib
- Grundkentnisse im Simulieren von Bränden

Verwendete Pakete und Datensätze

- NumPy
- pandas
- matplotlib
- fdsreader

Bearbeitungszeit

Geschätzte Bearbeitungszeit: 4h

Lernziele

- Einlesen von fds Daten mit dem fdsreader
- Analyse der Daten in Bezug auf ASET

1 Einführung in ASET

ASET (Available Safe Egress Time, auf Deutsch: Verfügbare Sichere Räumungszeit) ist ein zentrales Konzept im Brandschutzingenieurwesen. Es beschreibt die Zeitspanne, die den Gebäudenutzern zur Verfügung steht, um ein Gebäude sicher zu verlassen, bevor die Bedingungen durch Feuer, Rauch oder Hitze lebensbedrohlich werden. Die ASET wird anhand verschiedener Faktoren berechnet, darunter die Brandentwicklung, die Zeit bis zur Branderkennung sowie die baulichen Gegebenheiten des Gebäudes, wie Notausgänge und Löschanlagen.

Es ist entscheidend, dass die ASET größer ist als die erforderliche sichere Räumungszeit (RSET – Required Safe Egress Time), um effektive Evakuierungspläne zu erstellen und die Sicherheit der Personen im Gebäude im Notfall zu gewährleisten.

1.1 Datenerhebung

Die Daten, die wir hier betrachten, wurden mithilfe des Fire Dynamics Simulator (FDS) erzeugt. Der FDS (Fire Dynamics Simulator) ist ein Modell der numerischen Strömungsmechanik, das zur Simulation von feuergetriebenen Strömungen verwendet wird. Es ermöglicht die Analyse und Vorhersage des Brandverhaltens sowie dessen Auswirkungen auf Gebäude und Umgebungen.

> **Warnung**
>
> Dieses Lernmodul geht nicht weiter auf Simulationen oder den FDS ein. Die hier verwendeten Simulationsdaten werden als Download bereitgestellt.

2 Verfügbare sichere Evakuierungszeit (ASET)

```
import fdsreader
import matplotlib.pyplot as plt
import numpy as np
```

Dieses Beispiel demonstriert eine Analyse von Slice-Daten, um die Karte der **verfügbaren sicheren Evakuierungszeit (ASET)** sowie die zeitliche Entwicklung der Rauchsichtbarkeitsschicht zu bestimmen. Das verwendete Szenario ist eine Mehrraumwohnung.

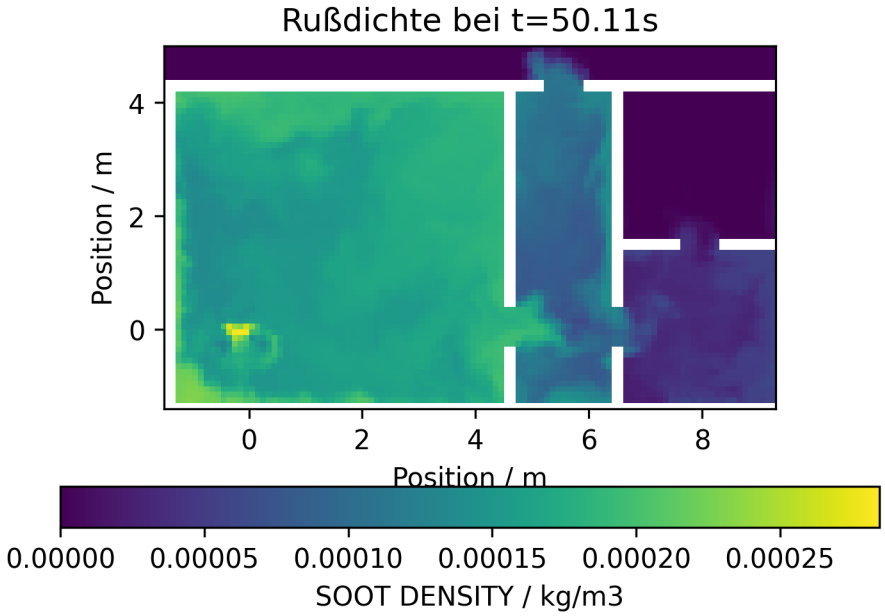
```
pfad_zur_simulation = '../skript/01-data/apartment_01'

sim = fdsreader.Simulation(pfad_zur_simulation)
print(sim)
```

```
Simulation(chid=Appartment,
           meshes=8,
           obstructions=23,
           slices=20,
           data_3d=5,
           smoke_3d=3)
```

```
# Rußdichte-Slice laden, senkrecht zur z-Achse in 1,5 m Höhe
slc = sim.slices.get_by_id('SootDensityZ_1.5m')

# Da die Simulation auf mehreren Gittern basiert, wird eine globale Datenstruktur erstellt;
# Wände werden als ungültige Datenpunkte (NaN) dargestellt
slc_data = slc.to_global(masked=True, fill=np.nan)
```

Zuerst wird eine Visualisierung der Daten zu einem ausgewählten Zeitpunkt mit der Funktion `imshow` durchgeführt.

```
# Zeitindex ermitteln
it = slc.get_nearest_timestep(50)

# Daten visualisieren
plt.imshow(slc_data[it,:,:].T, origin='lower', extent=slc.extent.as_list())

# Achsenbeschriftung und Farbleiste
plt.title(f'Rußdichte bei t={slc.times[it]:.2f}s')
plt.xlabel('Position / m')
plt.ylabel('Position / m')
plt.colorbar(orientation='horizontal', label=f'{slc.quantity.name} / {slc.quantity.unit}')
```



Nun werden die lokalen ASET-Werte berechnet:

1. Über alle räumlichen Elemente der Slice-Daten iterieren
2. Alle Zeitpunkte bestimmen, an denen der Begehbarkeits-Grenzwert überschritten wird
3. Falls dies der Fall ist, den ersten Zeitpunkt als lokalen ASET-Wert setzen

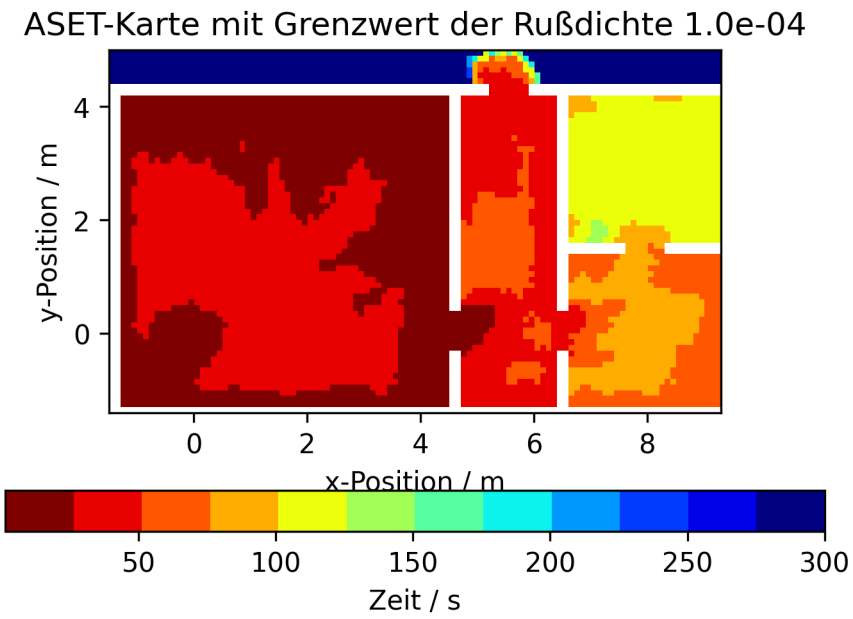
```
# Beliebiger Grenzwert für die Begehbarkeit
grenzwert_rußdichte = 1e-4

# Karte mit maximaler ASET als Standardwert erstellen
```

```
aset_karte = np.full_like(slc_data[0], slc.times[-1])

# Wände auf NaN setzen
aset_karte[np.isnan(slc_data[0,:,:])] = np.nan

# 1D-Schleife über alle Array-Indizes, ix ist ein zweidimensionaler Index
for ix in np.ndindex(aset_karte.shape):
    # Lokale Werte, die den Grenzwert überschreiten, ermitteln
    lokaler_aset = np.where(slc_data[:, ix[0], ix[1]] > grenzwert_rußdichte)[0]

    # Falls vorhanden, ersten Zeitpunkt als ASET-Wert verwenden
    if len(lokaler_aset) > 0:
        aset_karte[ix] = slc.times[lokaler_aset[0]]
```

Mit der berechneten Karte kann nun eine grafische Darstellung erzeugt werden, ähnlich wie bei anderen Größen – hier mit diskreter Farbskala.

```
# Diskrete (12 Werte) Farbskala erstellen
cmap = plt.cm.get_cmap('jet_r', 12)

# Visualisierung der ASET-Karte
plt.imshow(aset_karte.T, origin='lower', extent=slc.extent.as_list(), cmap=cmap)
plt.title(f'ASET-Karte mit Grenzwert der Rußdichte {grenzwert_rußdichte:.1e}')
plt.xlabel('x-Position / m')
plt.ylabel('y-Position / m')
plt.colorbar(orientation='horizontal', label='Zeit / s')

# Ausgabe in Datei speichern (optional)
# plt.savefig('figs/appartment_aset_map.svg', bbox_inches='tight')
# plt.close()
```

```
/tmp/ipykernel_3321/2604837150.py:2: MatplotlibDeprecationWarning: The get_cmap function was
  cmap = plt.cm.get_cmap('jet_r', 12)
```



2.1 Rauchsichtbarkeitsschicht

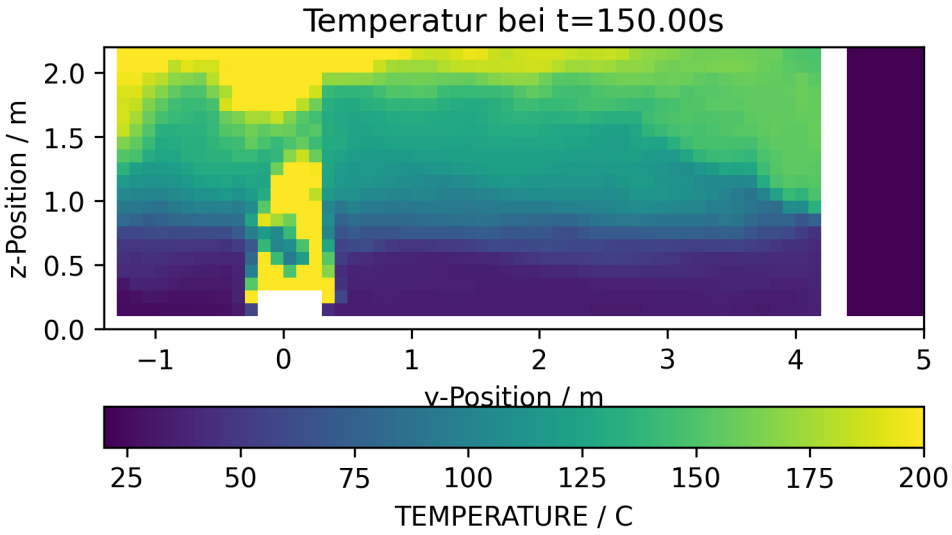
In diesem Beispiel wird die Höhe der Rauchsichtbarkeitsschicht analysiert. Die Unterscheidung erfolgt hier anhand eines einfachen Temperatur-Grenzwerts: Die lokale Schichthöhe ergibt sich aus dem niedrigsten Punkt, an dem eine bestimmte Temperatur überschritten wird. Die Auswertung erfolgt auf einem Slice entlang des Brenners (normal zur x-Achse).

```
# Slice finden
slc = sim.slices.get_by_id('BurnerTempX')

# In globale Datenstruktur umwandeln und Koordinaten extrahieren
slc_data, slc_coords = slc.to_global(masked=True, fill=np.nan, return_coordinates=True)
```

Zunächst erfolgt eine Visualisierung der Daten zu einem beliebigen Zeitpunkt. Weiße Bereiche stellen Hindernisse dar.

```
# Zeitpunkt wählen
it = slc.get_nearest_timestep(150)

# Daten visualisieren
plt.imshow(slc_data[it,:,:].T, origin='lower', vmax=200, extent=slc.extent.as_list())
plt.title(f'Temperatur bei t={slc.times[it]:.2f}s')
```

```
plt.xlabel('y-Position / m')
plt.ylabel('z-Position / m')
plt.colorbar(orientation='horizontal', label=f'{slc.quantity.name} / {slc.quantity.unit}')

# plt.savefig('figs/appartment_temp_slice.svg', bbox_inches='tight')
# plt.close()
```

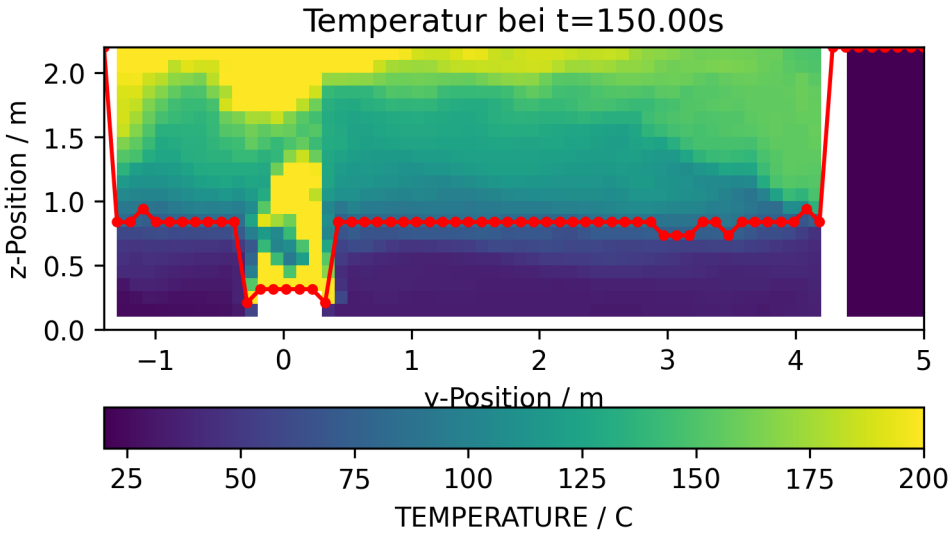


Nun wird für jede y-Position der z-Index gesucht, an dem die Temperatur den Grenzwert überschreitet. Der niedrigste dieser Punkte ist die lokale Höhe der Rauchsichtbarkeitsschicht.

```
# Grenzwert für Temperatur
temperatur_grenzwert = 75

# Array zur Speicherung der lokalen Höhenwerte; Standard ist maximale z-Koordinate
schicht_hoehe = np.full(slc_data.shape[1], slc_coords['z'][-1])

# Schleife über y-Indizes
for ix in range(len(schicht_hoehe)):
    # Indizes finden die den Grenzwert überschreiten
    lt = np.where(slc_data[it, ix, :] > temperatur_grenzwert)[0]
    # Wenn welche existieren, wähle den niedrigsten von ihnen
    if len(lt) > 0:
        schicht_hoehe[ix] = slc_coords['z'][lt[0]]
```

Die resultierenden Werte können nun über dem Slice dargestellt werden, um die Plausibilität zu überprüfen.

```
# Slice Daten
plt.imshow(slc_data[it,:,:].T, origin='lower', vmax=200, extent=slc.extent.as_list())
plt.title(f'Temperatur bei t={slc.times[it]:.2f}s')
plt.xlabel('y-Position / m')
plt.ylabel('z-Position / m')
plt.colorbar(orientation='horizontal', label=f'{slc.quantity.name} / {slc.quantity.unit}')
# Rauchsichtbarkeitsschichthöhe
plt.plot(slc_coords['y'], schicht_hoehe, '.-', color='red')

# plt.savefig('figs/appartment_temp_slice_height.svg', bbox_inches='tight')
# plt.close()
```



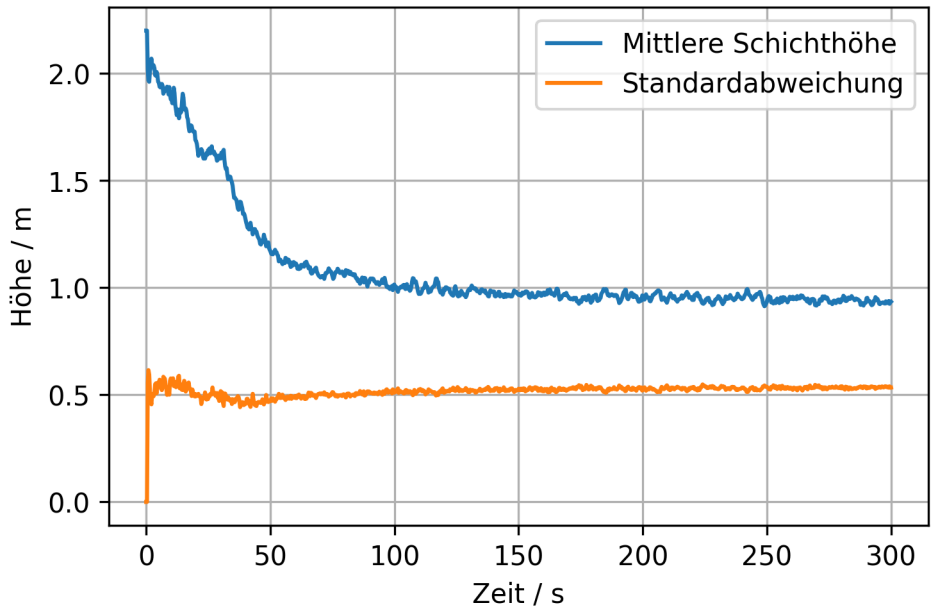
Die obige Methode kann auch über alle Zeitpunkte angewendet werden, um z. B. Mittelwert und Standardabweichung der Schichthöhe zu berechnen.

```
mittelwert = np.zeros_like(slc.times)
standardabweichung = np.zeros_like(slc.times)
res = np.zeros(slc_data.shape[1])

for it in range(len(slc.times)):
    res[:] = slc_coords['z'][-1]
    for ix in range(len(res)):
        lt = np.where(slc_data[it, ix, :] > temperatur_grenzwert)[0]
        if len(lt) > 0:
            res[ix] = slc_coords['z'][lt[0]]
```

```
    mittelwert[it] = np.mean(res)
    standardabweichung[it] = np.std(res)
```

```
# Darstellung des Mittelwerts und der Standartabweichung als Funktion der Zeit
plt.plot(slc.times, mittelwert, label='Mittlere Schichthöhe')
plt.plot(slc.times, standardabweichung, label='Standardabweichung')
plt.grid()
plt.legend()
plt.xlabel('Zeit / s')
plt.ylabel('Höhe / m')

# Ergebnisse in Datei abspeichern
# plt.savefig('figs/appartment_layer_mean_stddev.svg', bbox_inches='tight')
# plt.close()
```

```
Text(0, 0.5, 'Höhe / m')
```

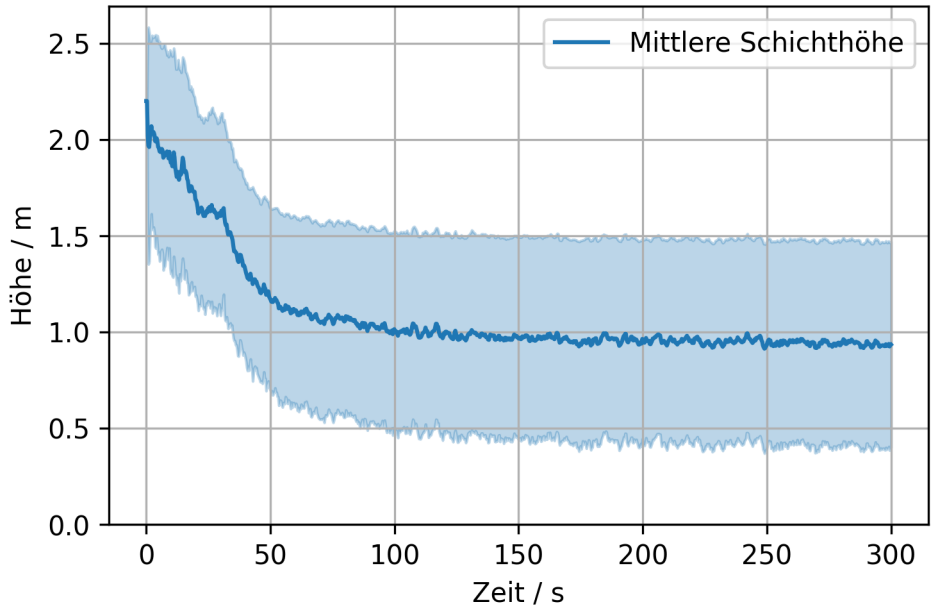


Beide Werte können kombiniert visualisiert werden. Die Standardabweichung ergibt ein Band um den Mittelwert.

```
# Mittelwert darstellen
plt.plot(slc.times, mittelwert, label='Mittlere Schichthöhe')
```

```
# Band um den Mittelwert darstellen mit Hilfe der Standartabweichung
plt.fill_between(slc.times, mittelwert-standardabweichung, mittelwert+standardabweichung, co

# Boden als Referenz zeigen
plt.ylim(bottom=0)
plt.grid()
plt.legend()
plt.xlabel('Zeit / s')
plt.ylabel('Höhe / m')

# plt.savefig('figs/appartment_layer_mean_band.svg', bbox_inches='tight')
# plt.close()
```

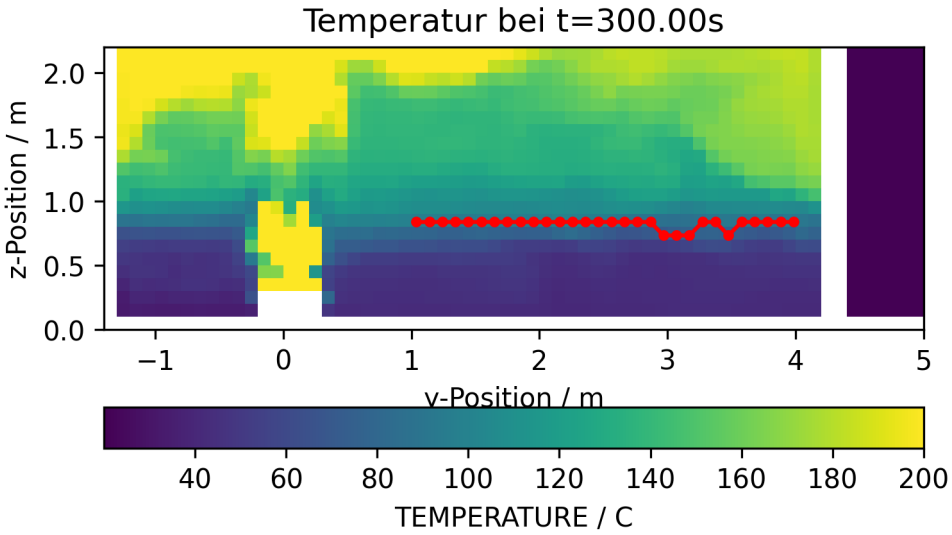
```
Text(0, 0.5, 'Höhe / m')
```



Wenn bestimmte Bereiche ausgeschlossen werden sollen, kann eine koordinatenabhängige Maske verwendet werden.

```
# Finde Indizes bei denen die y-Koordinate zwischen den angegebenen Werten liegt
ymin = 1
ymax = 4
koordinaten_maske = np.where((slc_coords['y'] > ymin) & (slc_coords['y'] < ymax))
```

```
# Slice Daten
plt.imshow(slc_data[it,:,:].T, origin='lower', vmax=200, extent=slc.extent.as_list())
plt.title(f'Temperatur bei t={slc.times[it]:.2f}s')
plt.xlabel('y-Position / m')
plt.ylabel('z-Position / m')
plt.colorbar(orientation='horizontal', label=f'{slc.quantity.name} / {slc.quantity.unit}')
# Rauchsichtbarkeitsschichthöhe
plt.plot(slc_coords['y'][koordinaten_maske], schicht_hoehe[koordinaten_maske], '.-', color='
# plt.savefig('figs/appartment_temp_slice_height_mask.svg', bbox_inches='tight')
# plt.close()
```

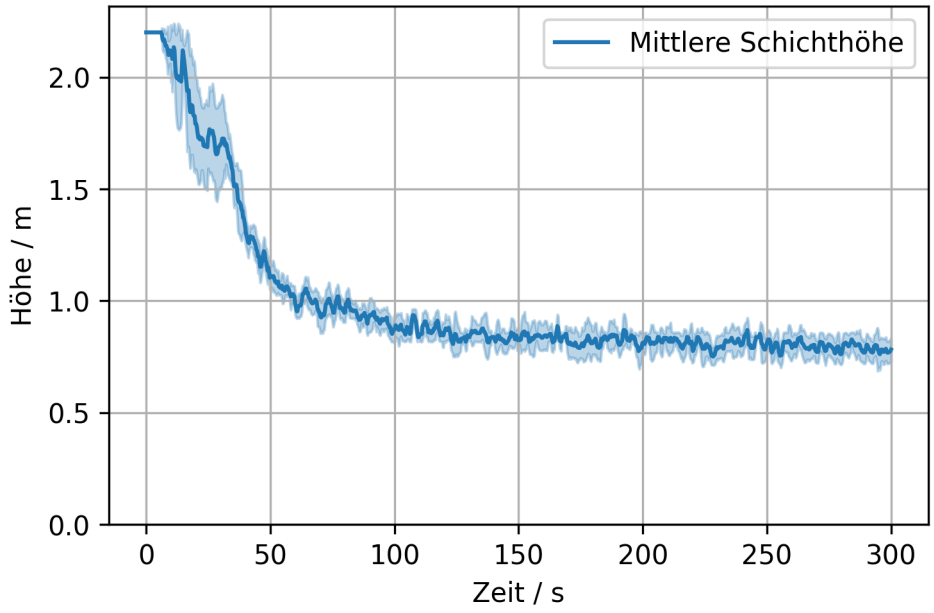


Das gleiche Verfahren kann erneut verwendet werden, wobei Mittelwert und Standardabweichung nur auf die maskierten Werte berechnet werden.

```
for it in range(len(slc.times)):
    res[:] = slc_coords['z'][-1]
    for ix in np.ndindex(res.shape):
        lt = np.where(slc_data[it, ix, :] > temperatur_grenzwert)[1]
        if len(lt) > 0:
            res[ix] = slc_coords['z'][lt[0]]
    # Berechnungen werden nun mit den Werten aus der Maske durchgeführt
    mittelwert[it] = np.mean(res[koordinaten_maske])
    standardabweichung[it] = np.std(res[koordinaten_maske])
```

```
# Selbe Darstellung wie zuvor
plt.plot(slc.times, mittelwert, label='Mittlere Schichthöhe')
plt.fill_between(slc.times, mittelwert-standardabweichung, mittelwert+standardabweichung, co
plt.ylim(bottom=0)
plt.grid()
plt.legend()
plt.xlabel('Zeit / s')
plt.ylabel('Höhe / m')

# plt.savefig('figs/appartment_layer_mean_band_mask.svg', bbox_inches='tight')
# plt.close()
```

```
Text(0, 0.5, 'Höhe / m')
```



3 Fdsreader

Um Simulationsdaten, die mit FDS berechnet wurden, mit Python auszuwerten, hat die Arbeitsgruppe von Prof. Lukas Arnold das Python-Modul `fdsreader` entwickelt. Ziel ist es, die meisten von FDS erzeugten Ausgabeformate auszulesen und in Python-Datenstrukturen abzubilden.

Das Modul ist frei verfügbar und Open Source. Der Quellcode ist auf GitHub gehostet: FireDynamics/fdsreader, eine API-Dokumentation ist ebenfalls verfügbar.

3.1 Installation und Import des Pakets

Das `fdsreader`-Modul kann über pip installiert werden (siehe auch das GitHub-Repository):

```
pip install fdsreader
```

Zur Einführung in die grundlegende Verwendung von `fdsreader` betrachten wir ein einfaches FDS-Szenario. Zunächst importieren wir das Modul:

```
import fdsreader
```

Da wir die Daten auch visualisieren möchten, importieren wir zusätzlich `matplotlib`:

```
import matplotlib.pyplot as plt
```

3.2 Auswahl des richtigen Ordners

Als Nächstes muss der Reader auf das Verzeichnis zeigen, das die Simulationsdaten enthält, insbesondere die Smokeview-Datei:

```
# Pfad zu den Daten definieren
path_to_data = '../skript/01-data/first_example'

sim = fdsreader.Simulation(path_to_data)
```

Das `Simulation`-Objekt `sim` enthält nun alle Informationen und Daten des Simulationsausgangs:

```
sim
```

```
Simulation(chid=StecklerExample,
           meshes=1,
           obstructions=7,
           slices=5,
           data_3d=5,
           smoke_3d=3,
           devices=4)
```

Die Variable `sim` enthält Informationen über das Gitter (`MESH`), vier Schnittdaten (`SLCF`) und vier Punktmessungen (`DEVC`). Das zusätzliche Messgerät – in der FDS-Eingabedatei wurden nur drei definiert – ist die Zeitspalte.

3.3 Messgeräte-Daten

> **Messgeräte in FDS**
>
> Messgeräte fungieren als virtuelle Sensoren, mit denen Daten wie Temperatur, Wärmestrom, Gaskonzentration, Geschwindigkeit usw. an bestimmten Punkten im Simulationsbereich aufgezeichnet werden können. Diese Daten sind entscheidend für das Verständnis des Verhaltens von Feuer und Rauch unter verschiedenen Bedingungen.
> Ein Messgerät kann mit einem Label (`ID`) versehen werden, was die Identifizierung in der durch FDS erzeugten CSV-Datei vereinfacht. Es benötigt eine Position und eine zu messende Größe.
> Positionen können auf verschiedene Arten angegeben werden. Wir fokussieren uns hier auf einen Punkt über `XYZ`. Linien, Flächen und Volumen sind jedoch ebenso möglich.
> Der Parameter `QUANTITY` erwartet eine Zeichenkette, die angibt, welche Größe aufgezeichnet werden soll, z. B. `TEMPERATURE` für die Gastemperatur.

Die einfachste Datenstruktur ist die Ausgabe der `DEVC`-Direktiven. Die verfügbaren Daten und Metainformationen können direkt ausgegeben werden:

```
# Kurzreferenz für Bequemlichkeit - `devc` enthält alle Messgeräte
devc = sim.devices
print(devc)
```

```
[Device(id='Time', xyz=(0.0, 0.0, 0.0), quantity=Quantity('TIME')),
Device(id='Temp_Door_Low', xyz=(1.45, 0.05, 0.1), quantity=Quantity('TEMPERATURE')),
Device(id='Temp_Door_Mid', xyz=(1.45, 0.05, 1.0), quantity=Quantity('TEMPERATURE')),
Device(id='Temp_Door_High', xyz=(1.45, 0.05, 1.65), quantity=Quantity('TEMPERATURE'))]
```

Die `Device`-Klasse enthält alle relevanten Informationen (siehe Geräte-Dokumentation):

```
for i in devc:
    print(f"ID: {i.id},\t Messgröße: {i.quantity_name}, \t Position: {i.position}")
```

```
ID: Time,    Messgröße: TIME,    Position: (0.0, 0.0, 0.0)
ID: Temp_Door_Low,   Messgröße: TEMPERATURE,     Position: (1.45, 0.05, 0.1)
ID: Temp_Door_Mid,   Messgröße: TEMPERATURE,     Position: (1.45, 0.05, 1.0)
ID: Temp_Door_High,  Messgröße: TEMPERATURE,     Position: (1.45, 0.05, 1.65)
```

Einzelne Messgeräte, einschließlich der Zeitspalte, sind über Dictionary-Einträge mit ihrer ID als Schlüssel zugänglich. Die Daten eines einzelnen Messgeräts (`Device.data`) sind als NumPy-Array gespeichert:

```
type(devc['Temp_Door_Mid'].data)
```

```
numpy.ndarray
```

Die Länge entspricht dem erwarteten Wert, also 1801, da die Simulation 1800 s dauerte und die Messgeräte jede Sekunde beschrieben wurden, einschließlich `t = 0 s`.

```
len(devc['Time'].data)
```

```
1801
```

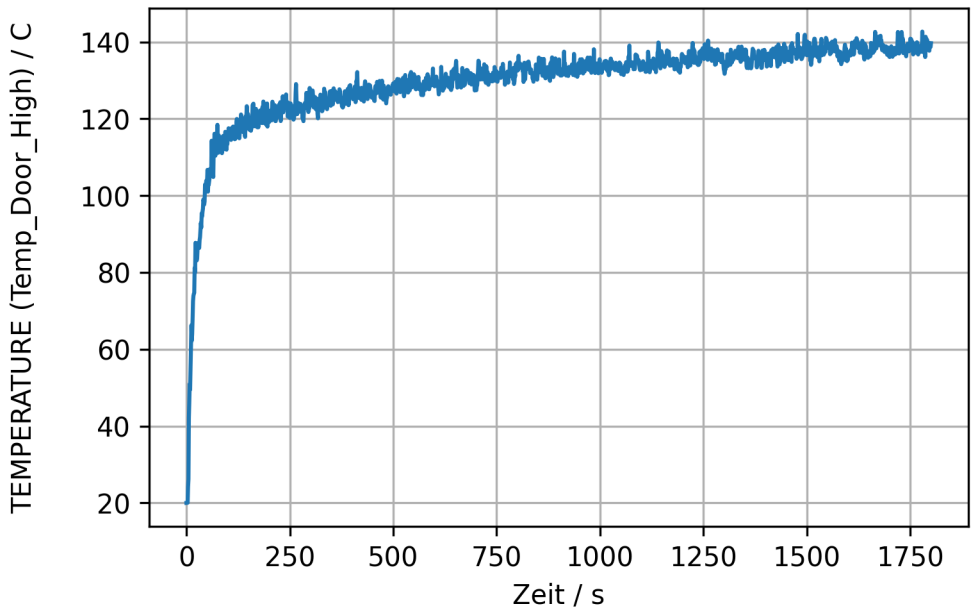
Ein erster Blick auf die Rohdaten (`Device.data`):

```
devc['Temp_Door_Mid'].data
```

```
array([ 20.      ,  20.002083,  20.034418, ..., 105.32822 , 114.82179 ,
       115.01705 ], shape=(1801,), dtype=float32)
```

Die Messgerät-Daten können auch mit Matplotlib visualisiert werden:

```
# Plot erstellen
plt.plot(devc['Time'].data, devc['Temp_Door_High'].data)

# Achsen beschriften
plt.xlabel("Zeit / s")
devc_id = devc['Temp_Door_High'].id
devc_q = devc['Temp_Door_High'].quantity_name
devc_u = devc['Temp_Door_High'].unit
plt.ylabel(f"{devc_q} ({devc_id}) / {devc_u}")

# Raster hinzufügen
plt.grid()
```

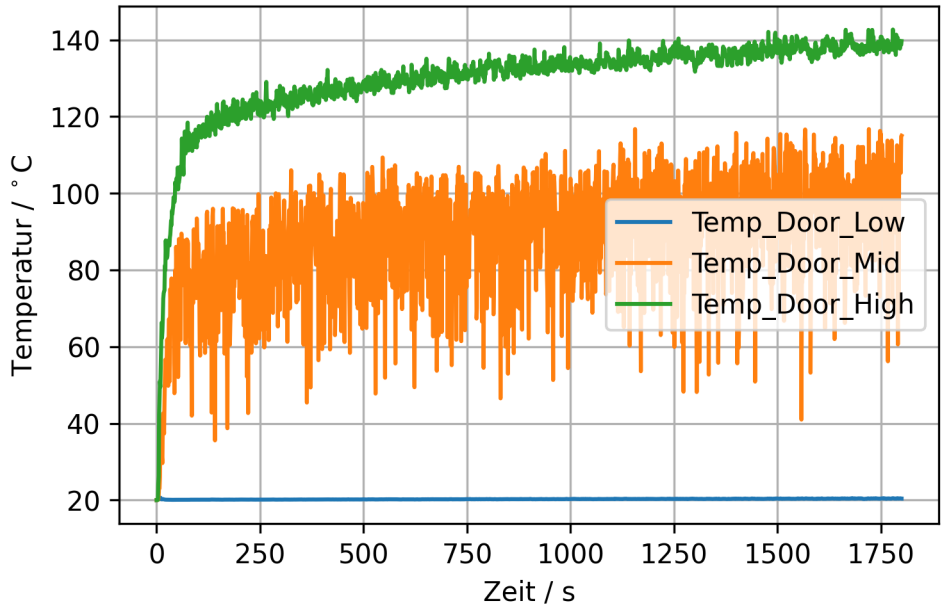


In gleicher Weise können mehrere Messgeräte gleichzeitig geplottet werden, z. B. alle, deren Namen mit `Temp_` beginnen:

```
# Alle Messgeräte durchlaufen
for i in devc:

    # Nur Messgeräte mit ID, die mit 'Temp_' beginnt
    if not i.id.startswith('Temp_'):
        continue

    plt.plot(devc["Time"].data, i.data, label=i.id)
```

```
plt.legend()
plt.xlabel("Zeit / s")
plt.ylabel('Temperatur / $^\circ$C')
plt.grid()
```

```
<>:12: SyntaxWarning: invalid escape sequence '\c'
<>:12: SyntaxWarning: invalid escape sequence '\c'
/tmp/ipykernel_3378/1963712746.py:12: SyntaxWarning: invalid escape sequence '\c'
  plt.ylabel('Temperatur / $^\circ$C')
```

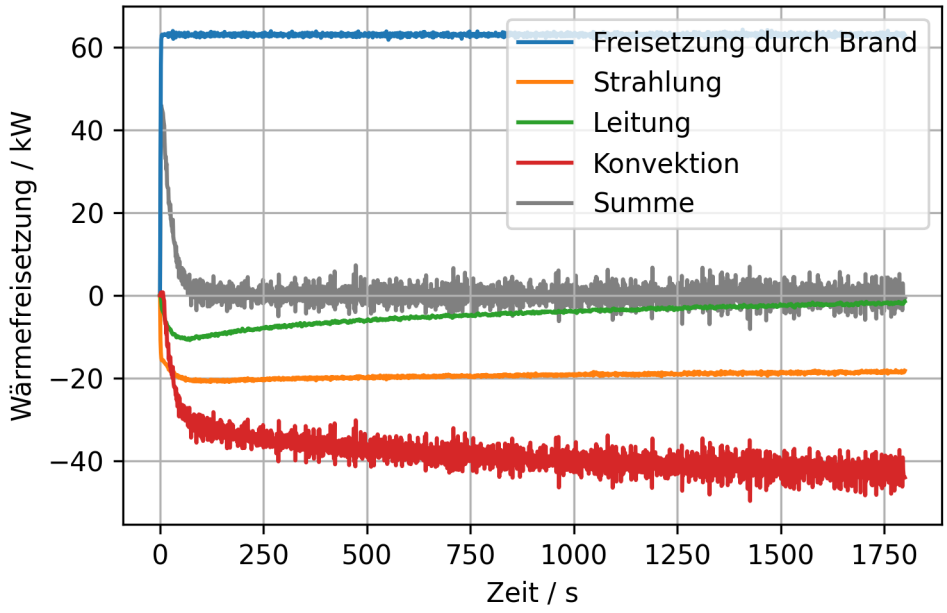


3.4 HRR-Daten

> **Heat Release Rate (HRR)**
>
> Ein zentraler Parameter in der Brandmodellierung. Er beschreibt die Freisetzungsrate thermischer Energie und wird in Kilowatt (kW) oder Megawatt (MW) angegeben.

Wie bei den `DEVC`-Daten kann auch auf die im `HRR`-File gespeicherten Daten direkt zugegriffen werden. Diese befinden sich im Attribut `hrr` des `Simulation`-Objekts und nicht bei den Messgeräten:

```
plt.plot(sim.hrr['Time'], sim.hrr['HRR'], label='Freisetzung durch Brand')

plt.plot(sim.hrr['Time'], sim.hrr['Q_RADI'], label='Strahlung')
plt.plot(sim.hrr['Time'], sim.hrr['Q_COND'], label='Leitung')
plt.plot(sim.hrr['Time'], sim.hrr['Q_CONV'], label='Konvektion')

plt.plot(sim.hrr['Time'],
         sim.hrr['HRR'] + sim.hrr['Q_RADI'] + sim.hrr['Q_COND'] + sim.hrr['Q_CONV'],
         color='grey', label='Summe', zorder=0)

plt.xlabel('Zeit / s')
plt.ylabel('Wärmefreisetzung / kW')
plt.legend()
plt.grid()
```



3.5 Slice-Daten

> **Slices**
>
> Slices sind eine Art der Ausgabe, bei der bestimmte physikalische Größen (z. B. Temperatur, Geschwindigkeit, Rauchkonzentration) innerhalb einer Ebene des Simulationsraums visualisiert werden können. Diese Schnitte geben einen Einblick, wie sich Größen über einen Bereich verteilen.

Die durch `SLCF` erzeugten Daten erstrecken sich über zwei oder drei räumliche Dimensionen sowie über die Zeit. Zusätzlich können sie auf mehrere Meshes verteilt sein.

Die Slice-Daten werden pro Mesh gespeichert. In diesem Beispiel gibt es nur ein Mesh, doch der Zugriff erfolgt trotzdem formell mit Index.

Die Datenstruktur sieht wie folgt aus:

```
sim.slices[sliceid][meshid].data[zeitindex, richtung1, richtung2]
```

Dabei ist `sliceid` der Index des Slices, `meshid` der Mesh-Index (hier: 0), und der Zugriff erfolgt über den Zeitindex sowie zwei Raumrichtungen (für 2D-Slices).

Es gibt mehrere Slice-Objekte:

```
# Verfügbare Slices ausgeben
for slice in sim.slices:
    print(f"Slicetyp [2D/3D]: {slice.type}\n  Größe: {slice.quantity.name}\n",
          f" Ausdehnung: {slice.extent}\n  Orientierung [1/2/3]: {slice.orientation}\n")
```

```
Slicetyp [2D/3D]: 2D
  Größe: TEMPERATURE
  Ausdehnung: Extent([0.00, 0.00] x [-1.40, 1.40] x [0.00, 2.20])
  Orientierung [1/2/3]: 1

Slicetyp [2D/3D]: 2D
  Größe: TEMPERATURE
  Ausdehnung: Extent([-1.40, 2.60] x [0.00, 0.00] x [0.00, 2.20])
  Orientierung [1/2/3]: 2

Slicetyp [2D/3D]: 2D
  Größe: W-VELOCITY
  Ausdehnung: Extent([0.00, 0.00] x [-1.40, 1.40] x [0.00, 2.20])
  Orientierung [1/2/3]: 1
```

```
Slicetyp [2D/3D]: 2D
  Größe: U-VELOCITY
  Ausdehnung: Extent([-1.40, 2.60] x [0.00, 0.00] x [0.00, 2.20])
  Orientierung [1/2/3]: 2

Slicetyp [2D/3D]: 2D
  Größe: W-VELOCITY
  Ausdehnung: Extent([-1.40, 2.60] x [-1.40, 1.40] x [1.80, 1.80])
  Orientierung [1/2/3]: 3
```

Es gibt viele Wege einen bestimmten Slice unter den anderen zu finden. Eine Möglichkeit, den gewünschten Slice zu finden, ist das Filtern nach Quantity über `filter_by_quantity`:

```python
# Slice(s) mit W-Geschwindigkeit bekommen
w_slice = sim.slices.filter_by_quantity("W-VELOCITY")
print(w_slice)
```

```
SliceCollection([Slice([2D] quantity=Quantity('W-VELOCITY'), cell_centered=False, extent=Exte
Slice([2D] quantity=Quantity('W-VELOCITY'), cell_centered=False, extent=Extent([-1.40, 2.60]
```

Oder die Auswahl über die Nähe zu einem Punkt:

```python
# Auswahl basierend auf der Nähe zu einem Punkt
slc = w_slice.get_nearest(x=1, z=2)
print(slc)
```

```
Slice([2D] quantity=Quantity('W-VELOCITY'), cell_centered=False, extent=Extent([-1.40, 2.60]
```

Der Zugriff auf Slice-Daten benötigt die Auswahl von einem bestimmten Mesh und Zeitindex. Die Funktion `get_nearest_timestep` hilft dabei:

```python
# Zeitindex nahe t=25 s auswählen
it = slc.get_nearest_timestep(25)
print(f"Zeitschritt: {it}")
print(f"Simulationszeit: {slc.times[it]}")
```

```
Zeitschritt: 25
Simulationszeit: 25.021108627319336
```

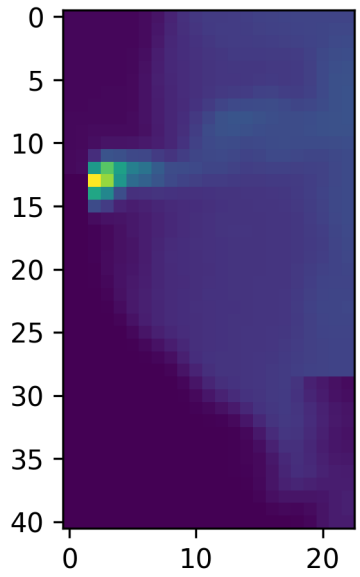
Das folgende Beispiel zeigt eine Darstellung der Daten und die benötigten Schritte um diese anzupassen. Die Anpassungen finden anhand der Datenausrichtung aus der Function `imshow` statt.

```
# Temperaturslice in y-Richtung auswählen
slc = sim.slices.filter_by_quantity('TEMPERATURE').get_nearest(x=3, y=0)
print(slc)
# Nur ein Mesh
slc_data = slc[0].data
print(slc_data)
```

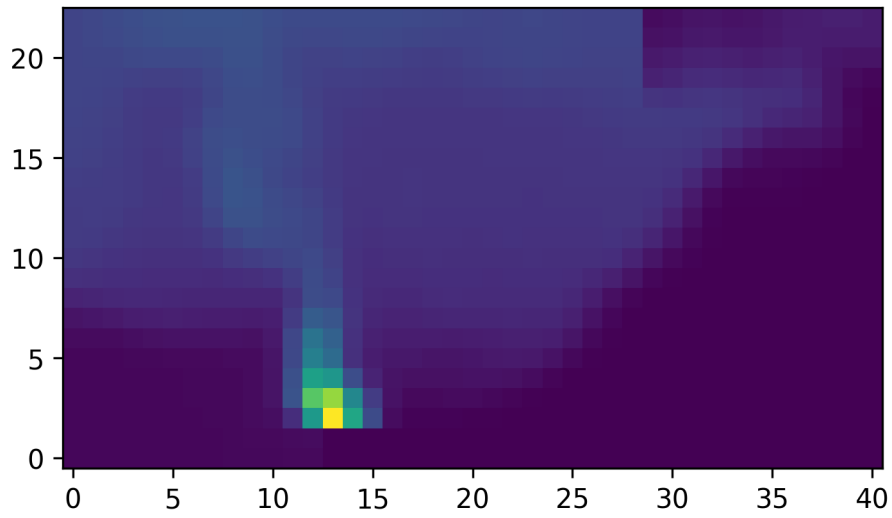
```
Slice([2D] quantity=Quantity('TEMPERATURE'), cell_centered=False, extent=Extent([-1.40, 2.60]
[[[ 20.        20.        20.       ...  20.        20.        20.      ]
  [ 20.        20.        20.       ...  20.        20.        20.      ]
  [ 20.        20.        20.       ...  20.        20.        20.      ]
  ...
  [ 20.        20.        20.       ...  20.        20.        20.      ]
  [ 20.        20.        20.       ...  20.        20.        20.      ]
  [ 20.        20.        20.       ...  20.        20.        20.      ]]

 [[ 20.030926  20.031328  20.032204 ...  20.001385  20.001268  20.00117 ]
  [ 20.030703  20.031597  20.033634 ...  20.001493  20.001345  20.001238]
  [ 20.031723  20.033785  20.038801 ...  20.001757  20.001535  20.001389]
  ...
  [ 20.006077  20.004908  20.002953 ...  20.001383  20.001154  20.00104 ]
  [ 20.005085  20.004053  20.00236  ...  20.00129   20.001116  20.001026]
  [ 20.004608  20.003656  20.0021   ...  20.00125   20.001104  20.001026]]

 [[ 20.12404   20.126698  20.133305 ...  20.026028  20.02525   20.025595]
  [ 20.116137  20.11882   20.12633  ...  20.02626   20.025606  20.02608 ]
  [ 20.114033  20.117645  20.128752 ...  20.02802   20.027351  20.027908]
  ...
  [ 20.018784  20.016739  20.013128 ...  20.00563   20.004776  20.004353]
  [ 20.015898  20.014067  20.010876 ...  20.005054  20.004427  20.004118]
  [ 20.01441   20.012737  20.00983  ...  20.004791  20.004278  20.00403 ]]

 ...

 [[ 44.00391   43.917053  43.920734 ... 143.89009  142.69537  142.16621 ]
  [ 44.004223  43.863914  43.708996 ... 143.29715  142.09953  141.6622  ]
  [ 43.81018   43.64982   43.4085   ... 142.64955  141.90448  141.75969 ]
  ...
```

```
 [ 20.284891  20.19156   20.076902 ...  90.631195  78.81051   72.00585 ]
 [ 20.218634  20.140545  20.047134 ...  56.04536   43.176456  39.645744]
 [ 20.151264  20.09307   20.028439 ...  34.67456   27.534237  27.970665]]

[[ 45.228874  45.115242  44.938766 ... 150.18481  150.12732  149.83371 ]
 [ 44.492287  44.350613  44.180614 ... 149.79759  150.0778   149.77635 ]
 [ 43.646873  43.590538  43.562504 ... 147.7298   148.82109  149.29768 ]
 ...
 [ 20.281096  20.186028  20.071451 ... 106.69953   93.09295   83.79199 ]
 [ 20.205025  20.13359   20.046276 ...  80.62758   71.11945   62.30358 ]
 [ 20.16152   20.102564  20.033293 ...  65.56552   56.724525  46.839134]]

[[ 42.762764  42.892406  42.67096  ... 146.0912   145.20709  144.58104 ]
 [ 43.14627   43.263447  43.141045 ... 145.02187  144.6713   143.69063 ]
 [ 43.753468  43.769325  43.798447 ... 141.0417   142.32797  141.77148 ]
 ...
 [ 20.268656  20.194078  20.08938  ...  72.89162   70.64532   65.348694]
 [ 20.206676  20.136755  20.052374 ...  59.554634  49.809177  42.573883]
 [ 20.180956  20.111738  20.035168 ...  48.16472   36.145966  31.134487]]]
```

```python
# Erste Visualisierung bei t=50 s
it = slc.get_nearest_timestep(50)
plt.imshow(slc_data[it])
```

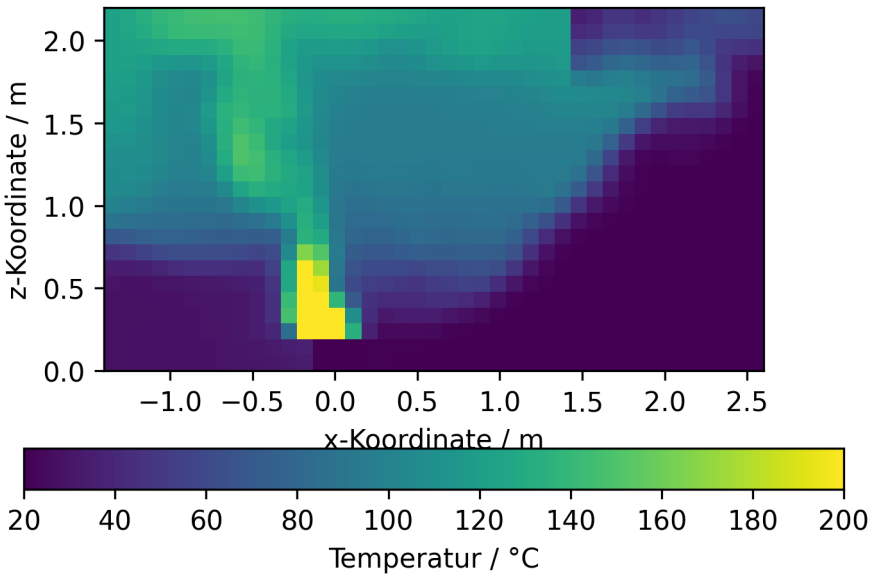


```
# Auf transponierte Darstellung mithilfe von ndarray.T zugreifen und anpassung des Ursprungs
plt.imshow(slc_data[it].T, origin='lower')
```

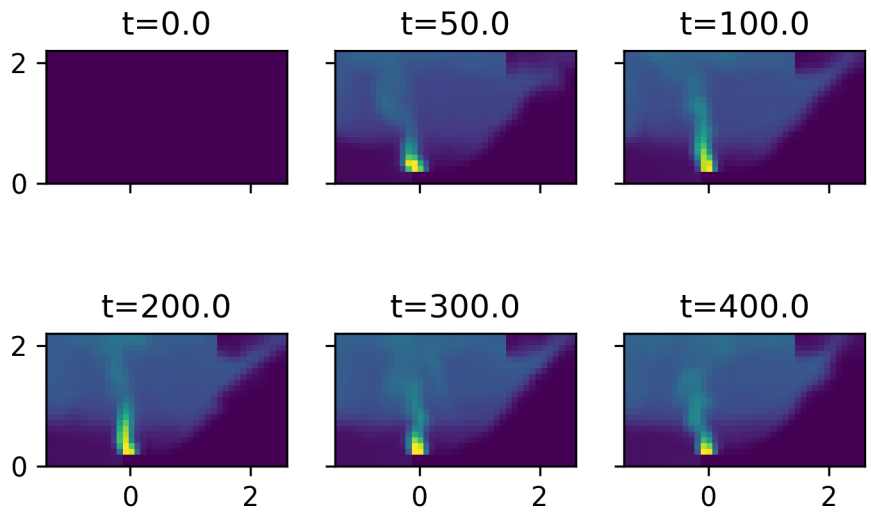



```
# Erste Visualisierung bei t=50 s
# Angabe der Ausdehnung um von Indexen in den physikalischen Raum zu wechseln
# Festsetzen eines Maximalwerts mit Hilfe von vmax

plt.imshow(slc_data[it].T,
           origin='lower',
           vmax=200,
           extent=slc.extent.as_list())
plt.colorbar(label='Temperatur / °C', orientation='horizontal')
plt.xlabel('x-Koordinate / m')
plt.ylabel('z-Koordinate / m')
```

```
Text(0, 0.5, 'z-Koordinate / m')
```




```
# Beispiel für Mehrfach-Plot
list_t = [0, 50, 100, 200, 300, 400]
fig, axs = plt.subplots(2,3, sharex=True, sharey=True)
for i in range(len(list_t)):
    it = slc.get_nearest_timestep(list_t[i])
    axs.flat[i].imshow(slc_data[it].T,
                       vmin=20,
                       vmax=400,
                       origin='lower',
                       extent=slc.extent.as_list())
    axs.flat[i].set_title(f"t={slc.times[it]:.1f}")
```


t=0.0
t=50.0
t=100.0
t=200.0
t=300.0
t=400.0
2
0
0
2

4

5