AUFGABE 1: Grundversion von miniTopSim

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

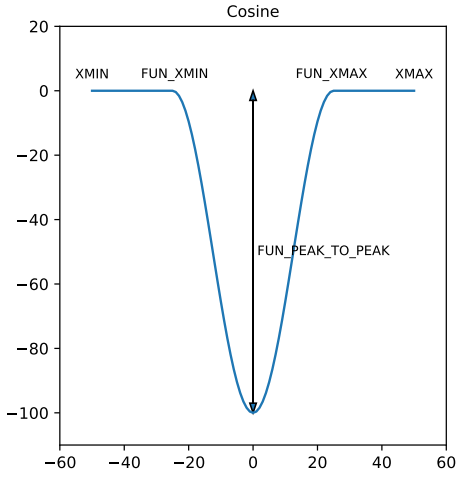
Motivation: Im Rahmen dieser Aufgabe soll die Grundstruktur des 2D Topographiesimulators miniTopSim festgelegt und die einfachstmögliche Simulation durchgeführt werden.

Aufgabenstellung: Die Oberfläche sei als Polygonzug definiert, d.h. als geradlinige Verbindung von Punkten $\vec{x}_i = (x_i, y_i)$, $i = 1 \ldots n$ (gedachte Einheit: nm). Die Punkte mögen sich iterativ mit Einheitsgeschwindigkeit $v_i = |\vec{v}_i| = 1$ [nm/s] (entsprechend isotropem Ätzen) entlang der jeweiligen Winkelsymmetralen der angrenzenden Segmente bewegen ($\vec{x}_i \to \vec{x}_i + \vec{v}_i \Delta t$). Die Zeit t läuft von $t = 0$ bis $t = t_{\text{end}}$ in Zeitschritten Δt. Die Simulationszeit t_{end} und die Zeitschrittweite Δt sollen zunächst als Argumente beim Aufruf von miniTopSim übergeben werden (`sys.argv` Variable). Später sollen sie aus dem *`cfg`*-File gelesen werden (siehe unten).

Die Oberfläche werde mit der Funktion

$$y = \begin{cases} -50 \cdot (1 + \cos \frac{2\pi x}{50}) & \text{if } |\mathrm{x}| < 25 \\ 0 & \text{otherwise} \end{cases}$$

im Bereich $-50 \leq x \leq 50$ mit einem Punktabstand in x-Richtung von $\Delta x = 1$ initialisiert. Die Funktion sieht wie nebenan dargestellt aus.



Schreiben Sie die Oberfläche vor dem ersten und nach jedem Zeitschritt in folgendem Format auf das File `basic_<tend>_<dt>.srf`, wobei im Filenamen `<tend>` durch t_{end} und `<dt>` durch Δt zu ersetzen ist:

```
surface:  time npoints x-positions y-positions
x[0] y[0]
x[1] y[1]
...
x[npoints-1] y[npoints-1]
```

Dabei ist „`time`" durch die aktuelle Zeit t, „`npoints`" durch die Anzahl n der Punkte der Oberfläche und „`x`" bzw. „`y`" durch die Punktkoordinaten zu ersetzen. „`surface:`" sowie „`x-positions`" und „`y-positions`" sind wörtlich zu schreiben. Weiters ist zu jedem Zeitschritt die aktuelle Zeit und die Zeitschrittweite auf Standard-Output zu schreiben.

Die Oberfläche ist zum Anfangs- und Endzeitpunkt graphisch darzustellen.

Lassen Sie den Simulator zweimal mit $t_{\text{end}} = 10\,\text{s}$ laufen, einmal mit $\Delta t = 10\,\text{s}$ und einmal mit $\Delta t = 1\,\text{s}$.

Arbeitsverzeichnis: *`miniTopSim/work/Aufgabe1_basic`*

Hier ist der Ort für *`*.cfg`*, *`*.srf`* und *`*.png`* Files. Von hier aus starten Sie miniTopSim entweder mittels *`python3 ../../minitopsim.py <tend> <dt>`* oder mit Hilfe des *`run.py`* Skripts.

Abzugebende Files: (*.py* files im *minitopsim* Package!)

main.py: Hauptmodul mit der `minitopsim()`-Funktion, die die Simulation startet.

surface.py: Modul, das eine Klasse `Surface` definiert, die folgende Methoden hat:

- `__init__(self, x, y)`: legt die beiden Attribute `self.x` und `self.y` an, die die Knoten der Oberfläche beschreiben. `x` und `y` sind array-like.
- `normal_vector(...)`: berechnet die in das Target hinein orientierten normierten Normalenvektoren an den Punkten (x,y) als Winkelsymmetralen der benachbarten Segmente. Verwenden Sie wenn möglich Whole-Array-Operationen.
- `plot(...)`: erzeugt einen Plot mit Hilfe von Matplotlib. Eine einfache Lösung reicht aus, da das Plotten Thema einer anderen Aufgabe ist.

io_surface.py: Modul, das die Funktion zur Berechnung der Anfangsbedingung enthält:

- `init_surface()`: implementiert die oben beschriebene Cosinus-Funktion, erzeugt damit eine `Surface` und gibt sie zurück.
- `write_surface(surface, time, srf_fobj)`: schreibt die Oberfläche auf das Fileobjekt `srf_fobj`.

advance.py: Modul, das die Bewegung der Oberfläche durchführt. Enthält die Funktionen

- `advance(surface, dtime)`: Funktion, die die Bewegung der Oberfläche für einen Zeitschritt der Größe `dtime` durchführt. `surface` ist das Oberflächenobjekt.
- `timestep(dt, time, tend)`: liefert den Zeitschritt zur Zeit `time`. `dt` ist die gewünschte Zeitschrittweite. Diese wird in der Regel unverändert zurückgegeben, außer wenn `time+dt` die Endzeit `tend` überschreitet. In diesem Fall wird `dt` entsprechend reduziert.

basic_10_10.srf, *basic_10_1.srf*: die Output-Files der beiden Simulationen.

basic_10_10.png, *basic_10_1.png*: die Plots für die beiden Simulationen. In jedem Plot soll die Oberfläche vor dem ersten und nach dem letzten Zeitschritt dargestellt sein. Markieren Sie die Stützstellen mit Symbolen.

Hinweis: Bei der Simulation entstehen Schleifen in der Oberfläche. Das Entfernen dieser Schleifen ist Thema einer anderen Aufgabe.

Postprocessing: Nachdem Ihr Kollege/Ihre Kollegin von AUFGABE 2 seine/ihre Arbeit vollendet hat (nach dem Vortrag), unterstützen Sie ihn/sie dabei, dass die Input-Parameter statt von der Kommandozeile vom *cfg*-File gelesen werden. miniTopSim soll dann mit *python3 ../../minitopsim.py etch.cfg* oder über ein *run.py* Skript im Arbeitsverzeichnis gestartet werden können. Die Parameter, die die Anfangsbedingung definieren (siehe Skizze auf erster Seite), sollen ebenfalls vom *cfg*-File gelesen werden. Weiters soll die `ETCH_RATE` vom *cfg*-File gelesen und bei der Simulation berücksichtigt werden. Das bei der Simulation generierte *srf*-File soll bis auf die Endung denselben Namen haben wie das *cfg*-File. Der Fall `ETCHING==False` braucht nicht implementiert zu werden. Das Einlesen von t_{end} und Δt von der Kommandozeile kann eliminiert werden.

Schreiben Sie ein *cfg*-File *etch.cfg*, welches $t_{\mathrm{end}} = 10$, $\Delta t = 1$ implementiert und verifizieren Sie, dass Sie dasselbe Ergebnis bekommen wie zuvor.

AUFGABE 2: Textbasierte Parametereingabe

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

Motivation: Python unterstützt das Lesen (und Schreiben) von Files in einem Format ähnlich dem von Microsoft Windows INI Files. Das Modul heißt `configparser`, die Files werden config- (*cfg*-)Files genannt. Diese erlauben das Gruppieren von Parametern in Sections, was insbesondere bei einer größeren Anzahl an Parametern sinnvoll ist. Ein *cfg*-File ist wie folgt formatiert:

```
[SectionName1]
ParameterName1 = Wert1
ParameterName2 = Wert2
...
[SectionName2]
ParameterNameN+1 = WertN+1
ParameterNameN+2 = WertN+2
...
```

`SectionName*` steht hier für den Namen einer Gruppe, `ParameterName*` für einen Parameternamen und `Wert*` für den zugehörigen Wert. Config-Files sind eine einfache und sinnvolle Möglichkeit, die Input-Parameter für eine Simulation anzugeben.

Weiters ist es sinnvoll, Default-Werte für die Parameter vorzusehen, sodass im *cfg*-File nur jene Parameter spezifiziert werden müssen, die von den Defaultwerten abweichen. Wir wollen diese in einer Parameterdatenbank (File *parameters.db*) abspeichern, die ebenfalls im *cfg*-Format geschrieben ist. Da wir weiters die Möglichkeit vorsehen wollen, zu jedem Parameter eine Bedingung und eine Erklärung abzuspeichern, ist der „Wert" der Parameter in der Datenbank ein Tupel:

```
[SectionName1]
ParameterName1 = (DefaultWert1, 'Bedingung1', '''Erklärung1''')
ParameterName2 = (DefaultWert2, 'Bedingung2', '''Erklärung2''')
...
[SectionName2]
ParameterNameN+1 = (DefaultWertN+1, 'BedingungN+1', '''ErklärungN+1''')
ParameterNameN+2 = (DefaultWertN+2, 'BedingungN+2', '''ErklärungN+2''')
...
```

Wir wollen auch die Möglichkeit vorsehen, dass ein Parameter im *cfg*-File obligatorisch spezifiziert werden muss. In diesem Fall wird in der Parameterdatenbank statt des Defaultwerts der Datentyp (`int`, `float`, `bool`) angegeben. Die Bedingung ist entweder `None` (d.h., es soll keine Bedingung überprüft werden) oder ein gültiger boolescher Python-Ausdruck, in einem String gespeichert. Die Erklärung könnte für eine Hilfefunktion in einem GUI oder für ein einfaches Manual verwendet werden.

Aufgabenstellung: Schreiben Sie eine Funktion im Modul *parameters.py*, die zunächst die Parameterdatenbank `parameters.db` liest und für jeden Parameter eine Modulvariable in *parameters.py* mit dem Defaultwert als Wert anlegt. Zweck der Modulvariablen ist, dass auf die Parameter von anderen Teilen des Programms nach

```
from .  import parameters as par
```

mit `par.ParameterName*` zugegriffen werden kann. Die Modulvariablen sollen großgeschrieben sein, unabhängig davon, wie sie im *cfg*-File und in der Parameterdatenbank geschrieben sind.

Dann soll das *cfg*-File gelesen werden, und für jeden Parameter soll überprüft werden,

- falls der Defaultwert ein Datentyp ist, ob der Parameter im *cfg*-File angegeben ist, sowie
- ob der Typ des Parameters im *cfg*-File mit dem Typ des Defaultwerts übereinstimmt, wobei ein `int`-Wert im *cfg*-File ohne Fehlermeldung in einen `float`-Wert umgewandelt werden soll, wenn in der Parameterdatenbank ein `float`-Wert steht.

Nachdem alle Parameterwerte in die Modulvariablen übertragen wurden, soll noch überprüft werden,

- ob die Bedingung aus der Parameterdatenbank erfüllt ist, sofern die Bedingung nicht `None` ist.

Die Fehler der Überprüfungen sollen als Fehlermeldungen ausgegeben werden. Falls es Fehler gibt, soll erst nach Ausgabe aller Fehlermeldungen das Programm abgebrochen werden.

Zum Testen schreiben Sie im Arbeitsverzeichnis ein Skript *load_parameters.py*, welches den Code aus *parameters.py* ausführt und die Modulvariablen mit ihren Werten auf ein File ausgibt. Das *cfg*-File soll als Argument auf der Kommandozeile angegeben werden (z.B. *python3 load_parameters.py test1.cfg*; das Output-File soll dann *test1.out* heißen). Führen Sie Tests mit mehreren *cfg*-Files durch einschließlich solcher, die zu einer Fehlerbedingung führen.

Arbeitsverzeichnis: *miniTopSim/work/Aufgabe2_param*

Abzugebende Files:

parameters.py: (im *minitopsim* Package) Enthält eine Funktion `load_parameters()`, die *cfg*-File und Parameterdatenbank (*parameters.db*) liest und hieraus Modulvariablen erzeugt. Benützen Sie dazu eine Funktion `parse_file()`, welche den Inhalt eines *cfg*-Files in ein Dictionary umwandelt.

load_parameters.py: Das Testprogramm (im Arbeitsverzeichnis), das den Namen des *cfg*-Files als Argument auf der Kommandozeile erhält, den Code aus *parameters.py* ausführt und die Parameter auf das *out*-File ausgibt.

**.cfg*, **.out*: *cfg*-Files und Output-Files der Tests.

parameters.db: unverändert.

Hinweis: Sie benötigen das `configparser` Modul sowie die Funktionen `type()`, `eval()`, `globals()` und `vars()`.

Postprocessing: Nachdem Ihre Kolleg_inn_en von AUFGABE 1 und 3 ihre Arbeit vollendet haben (nach dem Vortrag), implementieren Sie mit deren Hilfe in *main.py*, dass die Input-Parameter statt von der Commandline vom *cfg*-File gelesen werden und dass die Plotfunktion aufgerufen wird, wenn `PLOT_SURFACE==True` ist. Testen Sie mit geeigneten *cfg*-Files.

AUFGABE 3: Plotten der Oberfläche

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

Background: In miniTopSim ist die Oberfläche als Polygonzug definiert, d.h. als geradlinige Verbindung von Punkten $\vec{x}_i = (x_i, y_i)$, $i = 1 \ldots n$ (gedachte Einheit: nm). Diese Oberfläche wird als Funktion der Zeit berechnet und abgespeichert. miniTopSim schreibt die Oberfläche vor dem ersten und nach jedem Zeitschritt in folgendem Format auf ein *srf*-File,

```
surface:  time npoints x-positions y-positions
x[0] y[0]
x[1] y[1]
...
x[npoints-1] y[npoints-1]
```

Dabei ist „time" durch die aktuelle Zeit t, „npoints" durch die Anzahl n der Punkte der Oberfläche und „x" bzw. „y" durch die Punktkoordinaten zu ersetzen. „surface:" sowie „x-positions" und „y-positions" sind wörtlich zu schreiben. Da unser Simulator erst geschrieben wird, arbeiten Sie mit dem Beispielfile *trench.srf*, das sich bereits in Ihrem Arbeitsverzeichnis befindet.

Aufgabenstellung: Schreiben Sie ein Modul *plot.py* im `minitopsim` Package mit einer Funktion `plot(srf_file)`, welche die Oberflächen eines *srf*-Files als Polygone darstellt. Nach Aufruf der Funktion soll das File eingelesen und zunächst die erste Oberfläche angezeigt werden. Geben Sie Ihrer Funktion mit Hilfe eines Event-Loops (siehe Event Handling and Picking im Matplotlib User's Guide) folgende Funktionalität bei Drücken der angegebenen Tasten:

Leertaste: die nächste/vorhergehende Oberfläche anzeigen.

`'[0-9]'`: nur jede 2^n-te Oberfläche anzeigen, z.B. möge nach Drücken der Taste `'3'` bei Drücken der Leertaste nur jede 8-te Oberfläche angezeigt werden, also 7 Oberflächen übersprungen werden.

`'f'`: „first", die erste Oberfläche anzeigen.

`'l'`: „last", die letzte Oberfläche anzeigen.

`'r'`: „reverse", schaltet Leserichtung um (ob bei Drücken der Leertaste die nächste oder die vorhergehende Oberfläche angezeigt wird).

`'a'`: „aspect ratio", das Aspektverhältnis zwischen 1:1 (`'equal'`) und automatisch (`'auto'`) hin- und herschalten.

`'d'`: „delete", zwischen zwei Modi hin- und herschalten, in denen die Darstellung früherer Oberflächen beim Darstellen neuer Oberflächen gelöscht oder nicht gelöscht wird.

`'s'`: „save", den gegenwärtigen Plot auf ein *png*-File mit dem Namen des *srf*-Files schreiben, wobei die Endung *srf* gegen *png* getauscht wird.

`'b'`: „boundaries", zwischen zwei Modi hin- und herschalten, wo die Darstellungsgrenzen fix oder der neuen Oberfläche angepasst werden.

'q': „quit", Anwendung verlassen.

Eventuell müssen Sie einzelne rc-Parameter aus der keymap-Kategorie mittels `plt.rcParams['...'] = ...` setzen, um Default-Belegungen von Tasten (siehe *matplotlibrc* File) zu überschreiben.

Schreiben Sie weiters ein Skript, das ebenfalls *plot.py* heißt, sich aber im Projektverzeichnis *miniTopSim* befindet, welches die `plot()`-Funktion aus dem `plot`-Modul aufruft (Aufruf des Skripts aus dem Arbeitsverzeichnis: *python3 ../../plot.py trench.srf*).

Arbeitsverzeichnis: *miniTopSim/work/Aufgabe3_plot*

Abzugebende Files:

plot.py: enthält

- `plot(fname)`: Funktion, die aufgerufen werden kann, um den Plot zu starten.
- `Surface_Plotter`: Klasse, die die Funktionalität implementiert, mit den Methoden:
  - `__init__(self, srf_file)`: initialisiert `Surface_Plotter`-Objekt für ein *srf*-File.
  - `on_key_press(self, event)`: Callback-Funktion für key-press Events, die die zu ändernde Option für den Plot setzt und den Plot updatet.
  - `update_plot(self)`: Plottet den Plot mit den gewählten Optionen.
  - `run(self)`: startet Event-Loop.

io_surface.py: Modul, das eine Funktion zum Lesen einer Oberfläche vom *srf*-File enthält:

- `read_surface(srf_fobj)`: liest eine Oberfläche vom Fileobjekt `srf_fobj` und gibt sie als `Surface`-Objekt sowie die Zeit zurück.

surface.py: Modul, das eine Klasse `Surface` definiert, die folgende `__init__()`-Methode hat:

- `__init__(self, x, y)`: legt die Attribute `self.x` und `self.y` an, die die Knoten der Oberfläche beschreiben. `x` und `y` sind array-like.

plot.py: Skript (im *miniTopSim* Projektverzeichnis), das die `plot()`-Funktion aus dem `plot`-Modul aufruft.

trench.png: ein Bild, in dem jede zweite Oberfläche des Files *trench.srf* dargestellt ist.

trench.srf: unverändert.

Postprocessing: Nachdem Ihr Kollege/Ihre Kollegin von AUFGABE 2 seine/ihre Arbeit vollendet hat (nach dem Vortrag), unterstützen Sie ihn/sie dabei, dass die Plotfunktion von *main.py* aus aufgerufen wird, wenn `PLOT_SURFACE==True` ist. Testen Sie mit einem geeigneten *cfg*-File. Die Plot-Methode von `Surface` aus AUFGABE 1 wird dann nicht mehr benötigt und kann gelöscht werden.

AUFGABE 4: Sputtern

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

Background: Die Basisversion von miniTopSim implementiert isotropes Ätzen mit Einheitsgeschwindigkeit. Im Rahmen dieser Aufgabe soll der Code dahingehend erweitert werden, dass man alternativ Sputtern simulieren kann.

Trifft ein Ionenstrahl mit Strahlflussdichte F_{beam} (Atome/cm^{2}s) auf ein Oberflächenelement, so erzeugt dieser an einer Stelle, an der die (einwärts zeigende) Oberflächennormale mit der Strahlrichtung (negative y-Achse) den Winkel θ einschließt, eine Sputterflussdichte (Atome/cm^{2}s)

$$F_{\mathrm{sput}} = F_{\mathrm{beam}} \cdot Y_{\mathrm{s}}(\theta) \cdot \cos\theta.$$

$Y_{\mathrm{s}}(\theta)$ ist der Sputter Yield. Die Sputterflussdichte rechnet sich in die Normalengeschwindigkeit der Oberfläche über

$$v_{\perp} = F_{\mathrm{sput}}/N$$

um, wobei N die Atomdichte des Materials ist [cm^{-3}]. Die Geschwindigkeit soll in [nm/s] umgerechnet werden.

Aufgabenstellung: Implementieren Sie Sputtern in miniTopSim. Der Sputter-Yield soll nach der Yamamura Formel

$$Y = Y_0 \left(\cos\theta\right)^{-f} \exp\left[b\left(1 - \frac{1}{\cos\theta}\right)\right]$$

berechnet werden.

Ohne weitere Maßnahmen neigt der Code auch bei Entfernung der Loops (AUFGABE 5) zu Instabilitäten. Um diese zu vermeiden, sollen zwei Maßnahmen gesetzt werden:

1. An abgeschatteten Punkten (Punkte, oberhalb denen sich ein anderes Segment befindet) wird die Geschwindigkeit $v_{\perp} = 0$ gesetzt.
2. Nach jedem Zeitschritt wird überprüft, ob überhängende Strukturen entstanden sind. Wenn ja, wird der Zeitschritt verworfen und mit der halben Zeitschrittweite wiederholt. Nach einem akzeptierten Zeitschritt wird die Oberfläche auf die ursprünglichen x-Werte zurückinterpoliert.

Versuchen Sie, wenn möglich Whole-Array-Operationen zu verwenden. Das Entfernen von Schleifen ist nicht Gegenstand dieser Aufgabe. Führen Sie folgende Parameter in der Parameterdatenbank *`parameters.db`* und im Code ein:

`[Setup] ETCHING`: Flag, das angibt, ob isotrop geätzt (`ETCHING=True`) oder gesputtert (`ETCHING=False`) werden soll (Default: `False`).

`[Beam] BEAM_CURRENT_DENSITY`: Strahlstromdichte J (Einheit A/cm^2, Default: 0.001). Die Strahlflussdichte F_{beam} ergibt sich aus der Strahlstromdichte über $J = F_{\mathrm{beam}} \cdot e$ mit e der Elementarladung (im Modul `scipy.constants` definiert). Wir beschränken uns zunächst auf ortsunabhängige Strahlstromdichten.

`[Numerics] INTERPOLATION`: steuert, ob Maßnahme 2. gesetzt wird (Default: `False`).

`[Physics] SPUTTER_YIELD_0, SPUTTER_YIELD_F, SPUTTER_YIELD_B`: Die Parameter Y_0, f und b der Yamamura-Funktion (Default: Y_0=3.25, f=1.41, b=0.17).

`[Physics] DENSITY`: Atomdichte N (Einheit: Atome/cm^3, Default: `5e22` für Si).

Definieren Sie für jeden Parameter wenn sinnvoll eine Bedingung und schreiben Sie eine Erklärung.

Überprüfen Sie die Methoden zur Schattenbestimmung der Surface-Klasse mit einer geeigneten Oberfläche. Lassen Sie Ihren Code mit zwei mal zwei *cfg*-Files laufen, die eine Sputtertiefe ähnlich der Ätztiefe aus AUFGABE 1 erzeugen. Verwenden Sie einmal Ätzen und einmal Sputtern bzw. einmal die numerische Option 1 und einmal Option 2.

Arbeitsverzeichnis: *miniTopSim/work/Aufgabe4_sputter*

Abzugebende Files:

surface.py: In der `Surface`-Klasse sind folgende neue Methoden zu implementieren:

- `has_shadows(...)`: gibt ein Flag zurück, welches anzeigt, dass mindestens ein Punkt abgeschattet ist.
- `get_shadows(...)`: gibt zurück, welche Punkte abgeschattet sind.
- `interpolate(xnew)`: interpoliert die Oberfläche auf neue x-Werte.

advance.py: In diesem Modul ist eine neue Funktion `get_velocities()` zu implementieren:

- `advance(...)`: bewegt die Oberfläche.
- `get_velocities(...)`: berechnet die Oberflächengeschwindigkeit in Abhängigkeit von `[Setup] ETCHING` als Ätzrate oder aus dem Sputter-Yield. Im Fall des Sputterns wird der Winkel zwischen Strahlrichtung und Oberflächennormale berechnet und die Funktion `sputter_yield()` aus dem `sputtering`-Modul aufgerufen.
- `timestep(...)`: Schreiben Sie die Funktion so um, dass sie möglichst viel von der Timestep-Logik enthält.

Hinweis: Sie müssen das `sputtering`-Modul mit `from . import sputtering` und NICHT die `get_sputter_yield`-Funktion mit `from .sputtering import get_sputter_yield` importieren, da in letzterem Fall der Zustand von `get_sputter_yield` zum Zeitpunkt des Imports erhalten bleibt.

sputtering.py: In diesem Modul wird der Sputter Yield berechnet.

- `get_sputter_yield(...)`: berechnet den Sputter Yield nach Yamamura.

parameters.db: die modifizierte Parameterdatenbank.

etch[_interp].cfg, *yamamura[_interp].cfg*: *cfg*-Files, die die 2 Modi (Ätzen, Sputtern nach Yamamura) und die zwei numerischen Methoden testen.

**.png*: die zugehörigen Bilder, die den Sputterfortschritt zeigen.

check_shadows.py: Code, der die `has_shadows()` und `get_shadows()`-Methode überprüft.

check_shadows.png: Ein Surface-Plot, in dem die abgeschatteten Punkte markiert sind.

Postprocessing: Lassen Sie Ihre *cfg*-Files mit Delooping (AUFGABE 5) laufen, und kopieren Sie die *srf*-Files auf *srf_save*-Files.

AUFGABE 5: Delooping

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

Background: In miniTopSim ist die Oberfläche durch eine Folge von Punkten definiert, die durch lineare Segmente miteinander verbunden gedacht sind. Durch das Bewegen der Oberfläche können Schleifen entstehen, die entfernt werden sollen. Dies kann dadurch geschehen, dass für jede Kombination zweier Segmente bestimmt wird, ob es einen Schnittpunkt gibt, und wenn dies der Fall ist, die Schleife durch den Schnittpunkt ersetzt wird.

Das lineare Segment i zwischen den Punkten i und $i+1$ kann durch

$$\vec{x}(i,t) = \vec{x}_i + (\vec{x}_{i+1} - \vec{x}_i) \cdot t$$

bzw.

$$x(i,t) = x_i + (x_{i+1} - x_i) \cdot t$$

$$y(i,t) = y_i + (y_{i+1} - y_i) \cdot t$$

beschrieben werden, wobei $\vec{x}_i = (x_i, y_i)$ und $\vec{x}_{i+1} = (x_{i+1}, y_{i+1})$ die das Segment i begrenzenden Knoten (Endpunkte) sind und $0 \leq t < 1$ der Laufparameter ist. Die Bedingung für den Schnittpunkt zweier Segmente i und j ergibt sich aus $\vec{x}(i,s) = \vec{x}(j,t)$, wobei s den Laufparameter des ersten und t den Laufparameter des zweiten Segments bezeichnet. Dies ist ein System aus zwei Gleichungen, das nach s und t zu lösen ist. Ein Schnittpunkt liegt vor, wenn $0 \leq s < 1$ und $0 \leq t < 1$. Der Aufwand des Algorithmus ist $\mathcal{O}(n^2)$, wenn n die Anzahl der Knoten bezeichnet.

Aufgabenstellung: Implementieren Sie einen Algorithmus zur Detektion und Beseitigung von Schleifen. Dieser soll nach jedem Zeitschritt angewandt werden. Da der Aufwand $\mathcal{O}(n^2)$ ist, ist auf eine effiziente Programmierung mittels Whole-Array Operationen zu achten. Schließen Sie das Schneiden identer oder benachbarter Segmente durch Verwenden einer geeigneten Maske aus.

Hinweis: Um alle möglichen Paare von Punkten in einem Array zu betrachten, müssen Sie zunächst zweidimensionale Arrays für x und y konstruieren. Mit `xi, xj = np.meshgrid(x,x)` erhalten Sie zwei 2D Arrays, wobei die x-Werte sowohl in identen Zeilen von `xi` als auch in identen Spalten von `xj` abgespeichert sind. Wenn Sie z.B. die Differenz `xj-xi` bilden, dann enthält das erzeugte Array die Differenz aller Paare von x-Werten. Zur Lösung Ihres Problems müssen Sie $N \times N$ Systeme von 2 linearen Gleichungen lösen, wenn N die Anzahl der Segmente ist. Dies ist in einem einzigen Aufruf von `numpy.linalg.solve()` möglich. Außerdem könnten Sie die folgenden NumPy-Funktionen nützlich finden: `numpy.triu_indices()`, `numpy.linalg.det()`, `numpy.eye()`, `numyp.argwhere()`, sowie Boolesche Maskierung (Boolesche Arrays zur Indizierung).

Testen Sie Ihren Code mit einfachen Oberflächen, die Loops enthalten, sowie mit Hilfe des *cfg*-Files aus AUFGABE 1 (*etch.cfg*) mit verschiedenen Punktezahlen n, indem Sie `DELTA_X` variieren. Bestimmen Sie die Rechenzeit als Funktion von n, tragen Sie sie in eine Tabelle in Ihrem Arbeitsverzeichnis ein und erzeugen Sie einen Plot.

Arbeitsverzeichnis: *miniTopSim/work/Aufgabe5_deloop*

Abzugebende Files:

main.py: Erweitern Sie das Hauptprogramm, sodass am Ende der Simulation (vor Aufruf der Plot-Funktion) die Rechenzeit ausgegeben wird. Verwenden Sie dazu die `process_time()` Funktion.

surface.py: Erweitern Sie die `Surface` Klasse um eine `deloop()` Methode.

advance.py: Rufen Sie die `deloop()` Methode nach dem Zeitschritt auf.

check_deloop.py: Skript, das für Testoberflächen die Loops entfernt.

check_deloop.png*: Graphische Darstellung der Testoberflächen mit und ohne Loopentfernung.

etch.cfg: *cfg*-File aus AUFGABE 1.

etch.png: das zugehörige Bild.

timing.py: Plot-Skript für das Timing (Daten können hartkodiert sein).

timing.png: Plot mit den Rechenzeiten (doppeltlogarithmische Darstellung).

Postprocessing: Implementieren Sie mit den Oberflächen aus *check_deloop.py* in *test_deloop.py* Pytest-Unittests für das Delooping nach dem Muster von *test_advance.py* (AUFGABE 6).

AUFGABE 6: Unittests

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

Background: Automatisiertes Testen muss schnell gehen, damit man es auch oft tut. Häufiges Testen ist eine Voraussetzung für sauberen Code. Unittests sind Tests, die einzelne, möglichst kleine und voneinander isolierte Einheiten des Codes testen. Wir verwenden als Framework Pytest.

Aufgabenstellung: Schreiben Sie Unittests für das `parameters`-Modul, die Berechnung der Oberflächennormale und die `advance()`-Funktion:

1. Für die `load_parameters()`-Funktion des parameters-Modul schreiben Sie Tests unter Verwendung der *cfg*-Files aus AUFGABE 2. Überprüfen Sie, dass im Fall fehlerhafter Parameter dies erkannt wird und dass im Fall eines korrekten *cfg*-Files die Modulvariablen im parameters-Modul die richitgen Werte haben. Für den Fehlerfall ist es nützlich, wenn die `load_parameters()`-Funktion eine Exception auslöst. Wenn dies noch nicht so implementiert ist, ändern Sie das.

   Für das Testen im Fall, dass die Parameter fehlerfrei sind, ist es am bequemsten, wenn Sie das *cfg*-File und die Parameterdatenbank in Dictionaries umwandeln. Das wurde schon in AUFGABE 2 gemacht, Sie können den Code verwenden (dieser wird dann allerdings nicht getestet, da, wenn er fehlerhaft ist, der Fehler im Code wie im Test auftritt; diese Möglichkeit wollen wir ignorieren).

2. Parameterisieren Sie die Tests aus Punkt 1 (`@pytest.mark.parametrize()`), sodass Sie zwei parametrisierte Tests haben, einen für fehlerhafte *cfg*-Files und einen für fehlerfreie.

3. Scheiben Sie einen Test für die Berechnung der Oberflächennormalen. Initialisieren Sie hierzu eine Oberfläche mit den drei Punkten `(-1,0)`, `(0,0)` und `(1,2)`. Tun Sie das in einer Fixture (`@pytest.fixture()`).

4. Schreiben Sie einen Test für die Oberflächenbewegung, die in advance() berechnet wird, für einen Zeitschritt von 1 s und einer Ätzrate von 5 nm/s. Setzen Sie die nötigen Parameter des parameters-Modul direkt (d.h. ohne Aufruf von load_parameters()) in einer weiteren Fixture.

5. Markieren Sie alle Ihre Tests als Unittests, sodass Sie mit *pytest -m unittest* laufen und nicht mit *pytest -m regression*.

Verwenden Sie die Pytest Dokumentation (*https://docs.pytest.org/*).

Arbeitsverzeichnis: *miniTopSim/work/Aufgabe6_unittests*. Hier sollen die Testmodule abgelegt werden.

Abzugebende Files:

test_parameters.py: Test, der zwei parametrisierte Testfunktionen enthält, eine für fehlerhafte und eine für fehlerfreie cfg-Files.

test_advance.py: Test, der zwei Fixtures und zwei Testfunktionen enthält für des Testen des Normalenvektors und der Oberflächenbewegung.

AUFGABE 7: Strahlprofile

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

Background: In AUFGABE 4 wurde Sputtern mit einem homogenen Ionenstrahl implementiert. Die Strahlstromdichte J [A/cm^2] wird im Parameter `[Beam] BEAM_CURRENT_DENSITY` angegeben. Die Oberflächengeschwindigkeit $v_\perp$ [nm/s] wird über die Sputterflussdichte F_{sput} [Atome/cm^{2}s] aus der Strahlflussdichte $F_{\mathrm{beam}} = J/e$ [Atome/cm^{2}s] ausgerechnet:

$$v_\perp = 10^7 F_{\mathrm{sput}}/N, \qquad F_{\mathrm{sput}} = F_{\mathrm{beam}} \cdot Y_{\mathrm{s}}(\theta) \cdot \cos\theta$$

Dabei ist θ der Winkel zwischen der ins Materialinnere zeigenden Oberflächennormalen und der Strahlrichtung (negative y-Achse), $Y_{\mathrm{s}}(\theta)$ der Sputter Yield und N die Atomdichte [Atome/cm^3] (Parameter `[Physics] DENSITY`).

Aufgabenstellung: Implementieren Sie drei `Beam`-Klassen, welche Callable Objects definieren, welche die Strahlflussdichte F_{beam} [Atome/cm^{2}s] als Funktion des Ortes x berechnen, sowie eine Initialisierungsfunktion des `beam`-Moduls, welche das vom Benutzer gewünschte Modell auswählt:

1. Homogener Strahl („broad beam"): die Strahlflussdichte ist unabhängig vom Ort.

2. Gaußförmige Strahlverteilung:

$$F_{\mathrm{beam}}(x) = \frac{I/e}{\sqrt{2\pi}\sigma W_{\mathrm{z}}} \exp\left(-\frac{(x-x_{\mathrm{c}})^2}{2\sigma^2}\right)$$

   Hierin ist I der Strahlstrom in Ampere, e die Elementarladung, W_z die Strahl- bzw. Scan-Weite in der dritten (nicht simulierten) Koordinatenrichtung, x_{c} die Position des Strahlmittelpunkts und σ die Standardabweichung der Gaußfunktion. Statt der Standardabweichung ist es üblich, die FWHM (full width at half maximum) anzugeben, welche mit der Standardabweichung über $\mathrm{FWHM} = \sigma\sqrt{8\ln 2}$ zusammenhängt.

3. Strahlverteilung gemäß zweier Fehlerfunktionen (= Approximation eines Gaußförmigen Strahls, der schnell über ein Intervall $[x_1,x_2]$ gescannt wird):

$$F_{\mathrm{beam}}(x) = \frac{I/e}{2W_{\mathrm{x}}W_{\mathrm{z}}} \left\{ \mathrm{erf}\left(-\frac{x-x_2}{\sqrt{2}\sigma}\right) - \mathrm{erf}\left(-\frac{x-x_1}{\sqrt{2}\sigma}\right) \right\}$$

   Hierin sind $x_1 = x_{\mathrm{c}} - W_{\mathrm{x}}/2$, $x_2 = x_{\mathrm{c}} + W_{\mathrm{x}}/2$ die Grenzen des Scanintervalls (dort fällt die Strahlstromdichte auf die Hälfte des theoretischen Maximums ab). Im Vergleich zur Gaußförmigen Strahlverteilung tritt der zusätzliche Parameter W_x auf.

Führen Sie die folgenden Parameter ein:

`[Beam] BEAM_TYPE`: Modell des Strahlprofils (`'constant'`, `'Gaussian'` oder `'error function'`). Default: `'constant'`.

`[Beam] BEAM_CURRENT`: Strahlstrom I [A] für nicht-konstante Strahlprofile. Defaultwert: 1 nA. Für `BEAM_TYPE='constant'` soll nicht `BEAM_CURRENT`, sondern weiterhin `BEAM_CURRENT_DENSITY` verwendet werden.

[Beam] `SCAN_WIDTH`: Scanweite W_z [nm] in z-Richtung für nicht-konstante Strahlprofile. Default-Wert: 1 µm.

[Beam] `BEAM_CENTER`: Strahlmittelpunkt x_c für nicht-konstante Strahlprofile. Default: 0.

[Beam] `FWHM`: Full Width at half maximum (FWHM) [nm] für nicht-konstante Strahlprofile. Default: 100 nm.

[Beam] `ERF_BEAM_WIDTH`: Scanweite W_x [nm] in x-Richtung für Fehlerfunktionsprofile. Default: 1 µm.

Vergleichen Sie die 3 Strahlprofile graphisch. Verwenden Sie dabei für `BEAM_CURRENT_DENSITY` den Defaultwert und für `BEAM_CURRENT` Werte, die auf etwa dieselbe maximale Strahlflussdichte führen. Für die anderen Parameter verwenden Sie die Defaultwerte. Verwenden Sie diese `BEAM_CURRENT`-Werte im Folgenden weiter.

Schreiben Sie Unittests für die Strahlprofile, die die Flussdichte im Maximum und in einem geeigneten Abstand überprüfen.

Führen Sie schließlich für die drei Strahlprofile Simulationen durch, wobei Sie für den homogenen Strahl das cfg-File aus AUFGABE 4 übernehmen. Für den Gaußförmigen und Fehlerfunktions-Strahl wählen Sie als Anfangsbedingung eine ebene Oberfläche (`[Initial Conditions] FUN_PEAK_TO_PEAK=0`) und die Simulationszeit so, dass die entstehenden Gräben etwa gleich tief wie breit sind.

Arbeitsverzeichnis: *miniTopSim/work/Aufgabe7_beam*

Abzugebende Files:

main.py: Aufruf der Initialisierungsfunktion des `beam`-Moduls.

beam.py: Das neue `beam`-Modul mit der `Beam`-Klasse.

advance.py: Modifizierte `advance()` Funktion, die die Strahlstromdichte aus dem `beam`-Modul verwendet.

Hinweis: Sie müssen das `beam`-Modul mit `from . import beam` und NICHT die `beam_profile`-Funktion mit `from .beam import beam_profile` importieren, da in letzterem Fall der Zustand von `beam_profile` zum Zeitpunkt des Imports erhalten bleibt.

plot_beam.py: Code, der die drei Strahlprofile $F_\mathrm{beam}(x)$ vergleicht. Rufen Sie die Funktionen aus dem `beam`-Modul des *minitopsim*-Verzeichnisses auf und setzen Sie die vom `beam`-Modul benötigten Parameter des `parameters`-Modul von *plot_beam.py* aus, bevor Sie die `beam`-Funktionen/Methoden aufrufen.

plot_beam.png: der erzeugte Plot.

parameters.db: Die modifizierte Parameter-Datenbank

const.cfg, gauss.cfg, erf.cfg: *cfg*-Files, die jeweils das entsprechende Strahlprofil testen.

**.png*: Ergebnis-Plotfiles zu obigen *cfg*-Files.

test_beam.py: Unittests für die drei Strahlprofile.

AUFGABE 8: Redeposition

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

Background: In der vorliegenden Version von miniTopSim wird die Oberflächengeschwindigkeit $v_{\perp i}$ [nm/s] am Knoten i aus der Sputterflussdichte $F_{\mathrm{sput},i}$ [Atome/cm^{2}s] wie folgt berechnet:

$$v_{\perp,i} = 10^7 F_{\mathrm{sput},i}/N, \qquad F_{\mathrm{sput},i} = F_{\mathrm{beam},i} \cdot Y_{\mathrm{s}}(\theta_i) \cdot \cos\theta_i$$

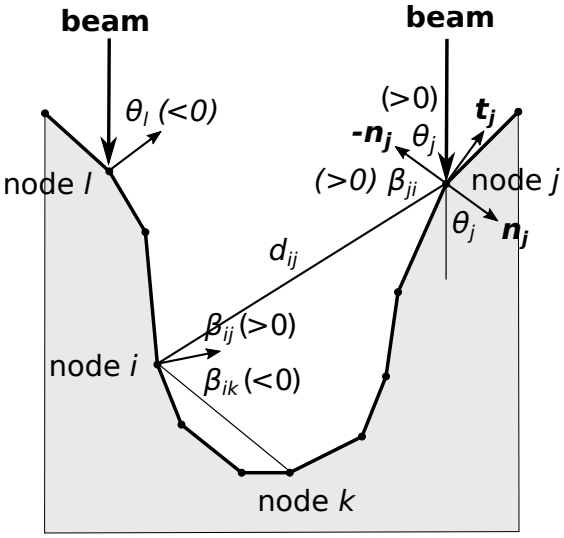
Die Sputterflussdichte ist die Anzahl der Targetatome, die pro Oberflächenelement und Sekunde am Knoten i abgetragen wird. θ_i bezeichnet den Winkel zwischen der Strahlrichtung (negative y-Achse) und der ins Materialinnere zeigenden Oberflächennormalen, die von Punkt zu Punkt variiert. In obiger Gleichung bedeuten weiters N die Atomdichte [Atome/cm^3] (Parameter `[Physics] DENSITY`), $F_{\mathrm{beam}} = J/e$ [Atome/cm^{2}s] die Strahlflussdichte, die über die Elementarladung e mit der Strahlstromdichte J [A/cm^2] zusammenhängt (Parameter `[Beam] BEAM_CURRENT_DENSITY`), und Y_{s} den Sputter-Yield, der im `sputtering`-Modul berechnet wird.

Hierbei wird angenommen, dass die gesputterten Atome ins Vakuum verschwinden und nicht an anderer Stelle der Probe redeponiert werden. Soll Redeposition berücksichtigt werden, so kann man für die vom Oberflächenelement Δl_j am Knoten j herrührende Redepositionsflussdichte $F_{\mathrm{redep},ij}$ am Knoten i

$$F_{\mathrm{redep},ij} = \frac{\cos\beta_{ij}\cos\beta_{ji}}{2d_{ij}} \cdot F_{\mathrm{sput},j}\Delta l_j$$

schreiben, wobei β_{ij} der Winkel zwischen der Verbindungslinie der zwei Knoten und der Oberflächennormalen am Punkt i, β_{ji} der entsprechende Winkel am Punkt j (vgl. Abbildung) und d_{ij} der Abstand der zwei Knoten ist. Δl_j ist die Länge des dem Knoten j zugeordneten Oberflächenelements, die durch den Mittelwert der Längen der benachbarten Segmente angenähert werden kann. Um die gesamte Redepositionsflussdichte $F_{\mathrm{redep},i}$ am Knoten i zu erhalten, muss man die Beiträge aller Knoten j aufsummieren:

$$F_{\mathrm{redep},i} = \sum_{j\,(\neq i)} F_{\mathrm{redep},ij}$$



Die Normalengeschwindigkeit des Knotens i ergibt sich zu

$$v_{\perp i} = 10^7(F_{\mathrm{sput},i} - F_{\mathrm{redep},i})/N$$

Eine wichtige Überlegung ist die Sichtbarkeit der Knoten j von Knoten i aus, denn die Redepositionsflüsse von nicht sichtbaren Knoten müssen 0 gesetzt werden. Eine notwendige Bedingung ist, dass $\beta_{ij} < 90°$ und $\beta_{ji} < 90°$ sind. In einfachen Fällen, auf die wir uns beschränken wollen, reichen diese Bedingungen aus.

Aufgabenstellung: Implementieren Sie Redeposition mit Hilfe der oben angeführten Formeln. Der Term

$$f_{ij} = \frac{\cos\beta_{ij}\cos\beta_{ji}}{2d_{ij}}\Delta l_j$$

heißt Viewfaktor.[1] Er definiert eine Matrix und soll in einer Methode der `Surface`-Klasse berechnet werden, da er eine Eigenschaft der Oberfläche ist. Die Redepositionsflussdichte $F_{\mathrm{redep},i}$ kann dann als Matrixmultiplikation der Viewfaktor-Matrix f_{ij} mit dem Vektor der Sputterflussdichten $F_{\mathrm{sput},j}$ berechnet werden. Dabei muss die Hauptdiagonale $f_{ii} = 0$ gesetzt werden. Sichtbarkeit berücksichtigen Sie, indem Sie Matrixelemente 0 setzen, für die $\cos\beta_{ij} \leq 0$ oder $\cos\beta_{ji} \leq 0$ ist. Die Matrixmultiplikation und die Berechnung der Geschwindigkeiten soll in der Funktion `get_velocities()` durchgeführt werden. Verwenden Sie soweit wie möglich Whole-Array-Operationen.

Damit Redeposition wahlweise berücksichtigt werden kann, führen Sie folgenden Parameter ein:

`[Setup] REDEP`: Flag, das angibt, ob Redeposition berücksichtigt werden soll. Der Defaultwert von `REDEP` ist `False`.

Führen Sie die Simulation (*yamamura_interp.cfg*) aus AUFGABE 4 mit und ohne Redeposition aus und vergleichen Sie die Ergebnisse. Implementieren Sie weiters einen Test, der nur einen Zeitschritt der Simulation mit Redeposition durchführt. Kopieren Sie das *srf*-File auf ein File mit Endung *srf_save*-File und testen Sie, ob bei einem erneuten Lauf das *srf*-File ident mit dem *srf_save*-File ist.

Arbeitsverzeichnis: *miniTopSim/work/Aufgabe8_redep*

Abzugebende Files:

surface.py: Das erweiterte `surface`-Modul.

advance.py: mit der modifizierten `get_velocities()` Funktion.

parameters.db: Die erweiterte Parameter-Datenbank.

yamamura_interp.cfg, yamamura_interp_redep.cfg: *cfg*-Files für die Wiederholung der Simulation aus AUFGABE 4, jetzt jedoch zusätzlich unter Berücksichtigung der Redeposition.

yamamura_interp.png, *yamamura_interp_redep.png*: Plotfiles zu obigen *cfg*-Files.

yamamura_interp_redep_1.cfg: wie *yamamura_interp_redep.cfg*, jedoch mit nur einem Zeitschritt.

test_interp_redep_1.py: Test, der *yamamura_interp_redep_1.srf* mit *yamamura_interp_redep_1.srf_save* vergleicht.

Postprocessing:

Plotten Sie die Kurven aus *yamamura_interp.png* und *yamamura_interp_redep.png* in einen Plot.

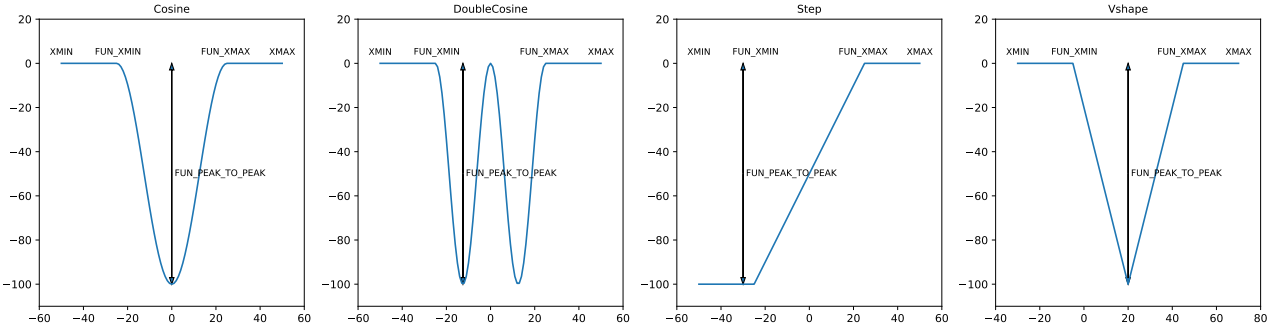
[1] In der gebräuchlichen Definition des Viewfaktors ist Δl_j durch Δl_i ersetzt, für unsere Anwendung ist jedoch diese Definition sinnvoll.

AUFGABE 9: Anfangsbedingungen und Plotten zweier Oberflächen

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

Background: Um miniTopSim gründlicher testen zu können, sollen weitere Anfangsbedingungen implementiert werden, und es soll die Möglichkeit geschaffen werden, zwei Oberflächen miteinander graphisch zu vergleichen.

Aufgabenstellung (Anfangsbedingungen): Implementieren Sie als weitere Optionen zur Initialisierung der Oberfläche eine flache Oberfläche ($y = 0$), eine der im Bild dargestellten analytischen Funktionen, sowie die Möglichkeit zum Einlesen eines *srf*-Files.



Die in den Abbildungen ersichtlichen Parameter sollten bereits in der Parameterdatenbank eingeführt sein. Für die dargestellten Oberflachen ist `FUN_PEAK_TO_PEAK=-100`. Um die Funktion auszuwählen, führen Sie den folgenden Parameter ein:

`[Initial Conditions] INITIAL_SURFACE_TYPE`: Funktionstyp (Default: `'Cosine'`)

- `INITIAL_SURFACE_TYPE='Flat'`: Flache Oberfläche $y = 0$.
- `INITIAL_SURFACE_TYPE='Cosine'`: Cosinusfunktion wie bereits implementiert.
- `INITIAL_SURFACE_TYPE='DoubleCosine'`: wie `'Cosine'` jedoch mit zwei Perioden.
- `INITIAL_SURFACE_TYPE='Step'`: Stufenfunktion mit Neigungswinkel.
- `INITIAL_SURFACE_TYPE='V-Shape'`: Symmetrisches V-Profil.
- `INITIAL_SURFACE_TYPE='File'`: Einlesen von einem *srf*-File.

Im Falle von `INITIAL_SURFACE_TYPE='File'` muss zusätzlich noch der folgende Parameter angegeben sein:

`[Initial_Conditions] INITIAL_SURFACE_FILE`: Filename des einzulesenden *srf*-Files. Wird nur verwendet, wenn `INITIAL_SURFACE_TYPE='File'`.

Testen Sie die neuen Anfangsbedingungen mit Hilfe von *cfg*-Files ähnlich jenem aus AUFGABE 1, jedoch mit `FUN_XMIN=-30`, `FUN_XMAX=10`. Achten Sie darauf, dass die Oberfläche keine Unstetigkeiten hat. Führen Sie eine der Simulationen zusätzlich zweimal mit der halben Simulationszeit aus, wobei die zweite Simulation das Ergebnis der ersten Simulation einliest.

Aufgabenstellung (Plotten zweier Oberflächen): Erweitern Sie die Plotfunktion dahingehend, dass die Oberflächen aus zwei *srf*-Files gleichzeitig dargestellt werden können, davon die zweite strichliert. Die Auswahl der ersten Oberflächen soll wie bisher innerhalb der Plotfunktion erfolgen. Aus dem zweiten File soll jeweils jene Oberfläche dargestellt werden, deren Simulationszeit jener aus dem ersten File am ähnlichsten ist. Simulieren Sie das Beispiel aus AUFGABE 1 mit `TIME_STEP=1 (wie bisher)` und `TIME_STEP=0.5` und stellen Sie die Oberflächen beider Simulationen bei $t = 9.5s$ in einem Plot dar.

Modifizieren Sie miniTopSim so, dass am Ende überprüft wird, ob ein File mit gleichem Filenamen aber Endung *srf_save* existiert und gegebenenfalls die Plotfunktion mit diesem als zweitem Parameter aufgerufen wird. Lassen Sie mit diesem das Beispiel AUFGABE 1 laufen, wobei Sie das entsprechende *srf_save*-File zuvor in einem identen Lauf erzeugt haben. Stellen Sie sicher, dass beide Oberflächen sichtbar sind, auch wenn sie ident sind.

Arbeitsverzeichnis: *miniTopSim/work/Aufgabe9_initial*

Abzugebende Files:

init_surface.py: Code zum Initialisieren der Oberfläche.

main.py: überprüft, ob ein *cfg_save*-File existiert und ruft gegebenenfalls die Plotfunktion mit diesem als weiterem Parameter auf.

plot.py: Die modifizierte Plotfunktion und das modifizierte Plotskript.

parameters.db: Die modifizierte Parameter-Datenbank

flat.cfg, cosine.cfg, double_cosine.cfg, step.cfg, vshape.cfg: *cfg*-Files, die jeweils die entsprechende Oberflächeninitialisierung testen.

flat.png, cosine.png, *double_cosine.png*, *step.png*, *vshape.png*: Die zugehörigen Ergebnis-Plots.

cosine_dt0_5.cfg: *wie cosine.cfg*, jedoch `TIME_STEP=0.5`.

cosine_dt0_5.png: Plot, der die Ergebnisse *cosine_dt0_5.srf* und *cosine.srf* bei $t = 9.5s$ vergleicht.

cosine_1.cfg, *cosine_2.cfg*: Wie *cosine.cfg*, jedoch mit der halben `TOTAL_TIME`. Im zweiten File wird die letzte Oberfläche vom File *cosine_1.srf* als Anfangsbedingung eingelesen.

cosine_2.png: Plot, der die letzten Oberflächen von *cosine_2.srf* und *cosine.srf* vergleicht.

AUFGABE 10: Adaptives Gitter

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

Background: Im Laufe einer Simulation können sich benachbarte Knoten erheblich voneinander entfernen bzw. einander nähern. Weiters können an Knicken in der Oberfläche Schleifen entstehen (wurden in AUFGABE 5 behandelt) oder runde, auseinanderlaufende Fronten (sogenannte Rarefaction Waves) bilden. Das erste Problem lässt sich durch Löschen oder Einfügen von Knoten beheben, die Rarefaction Waves durch Einfügen von Punkten mit identen Koordinaten und unterschiedlichen Oberflächennormalen.

Aufgabenstellung: Implementieren Sie eine `adapt()`-Methode in der `Surface`-Klasse, die sowohl ein Längen- als auch ein Winkelkriterium heranzieht, um neue Punkte einzufügen oder alte Punkte zu entfernen:

1. Implementieren Sie einen Algorithmus, der die Segmentlänge näherungsweise konstant hält: Überschreitet die Segmentlänge den Wert $d_{\max}$, dann soll ein Punkt eingefügt werden. Wird der Abstand zwischen linkem und rechtem Nachbarpunkt eines Punktes kleiner als $d_{\max}$, dann soll der Punkt entfernt werden.

   Für das Einfügen von Punkten muss die Oberfläche zwischen den alten Knoten interpoliert werden. Benützen Sie dazu die `interp1d` Klasse aus SciPy. Vergleichen Sie Ergebnisse mit linearer, quadratischer und kubischer Interpolation. Die `interp1d` Klasse setzt voraus, dass die x-Werte aufsteigend geordnet sind. Ist dies nicht gegeben, soll die Simulation abgebrochen werden.

2. Ist der Winkel zwischen Oberflächennormalen benachbarter *Segmente*[2] größer als $\alpha_{\max}$, so sollen zusätzliche Knoten mit identen Koordinaten, aber unterschiedlichen Oberflächennormalen eingefügt werden. Im Gegensatz zum Längenkriterium sollen auf Grund des Winkelkriteriums keine Knoten entfernt werden. Weiters soll das Entfernen von Punkten auf Grund des Abstandskriteriums unterdrückt werden, wenn dadurch das Winkelkriterium verletzt würde.

Rufen Sie die `adapt()`-Methode an geeigneter Stelle in der `advance()`-Funktion auf.

Führen Sie die folgenden Parameter ein:

`[Numerics] ADAPTIVE_GRID`: Flag, das angibt, ob das Gitter adaptiert werden soll (Defaultwert: `True`).

`[Numerics] MAX_SEGLEN`: Maximaler Abstand $d_{\max}$ benachbarter Knoten (Defaultwert: `2.`).

`[Numerics] MAX_ANGLE`: Maximaler Winkel $\alpha_{\max}$ zwischen benachbarten Oberflächennormalen (Default: `10°`).

Schreiben Sie einige Unittests für sehr einfache Oberflächen, die nur aus einigen wenigen Punkten bestehen. Überprüfen Sie für die in AUFGABE 9 implementierten Anfangsbedingungen (Parameter `[Initial Conditions] TYPE`) die Auswirkungen des adaptiven Gitters. Schreiben Sie weiters einen Test, der die Simulation mit der Cosinusfunktion als Anfangsbedingung mit dem gespeicherten Resultat vergleicht. Optional (d.h., wenn Sie noch Lust haben): Schalten Sie die Redeposition ein.

[2] nicht: „benachbarter *Punkte*"!

Arbeitsverzeichnis: *miniTopSim/work/Aufgabe10_adapt*

Abzugebende Files:

surface.py: Um die `adapt()`-Methode erweiterte `Surface`-Klasse.

advance.py: Um den Aufruf von `adapt()` erweiterte `advance()`-Funktion.

parameters.db: Die erweiterte Parameter-Datenbank.

test_adapt.py: Unittest für die `adapt()`-Methode.

cosine.cfg, double_cosine.cfg, step.cfg, vshape.cfg: *cfg*-Files wie in AUFGABE 9, jedoch mit aktiviertem adaptiven Gitter. Simulieren Sie hinreichend lange Zeiten, sodass alle Features sichtbar werden.

**.png*: Die zugehörigen Bilder, die die Oberflächen am Ende der Simulation darstellen.

test_cosine.py: Test, der *cosine.srf* mit *cosine.srf_save* vergleicht.

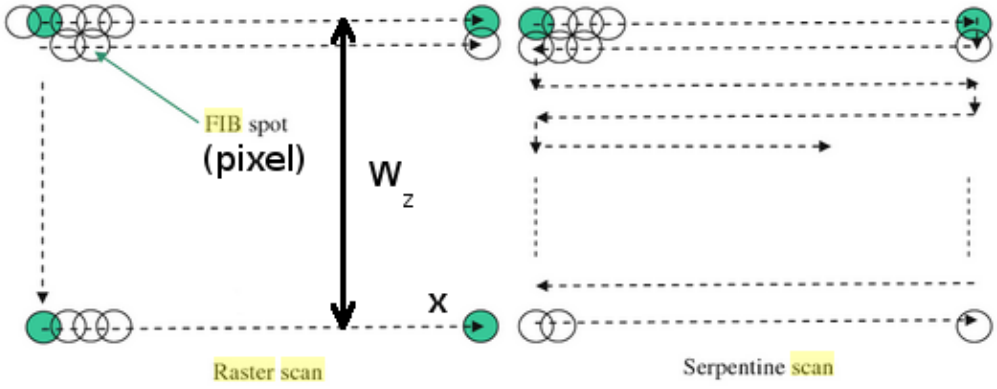
AUFGABE 11: Gescannter Strahl

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

Background: In AUFGABE 7 wurde unter anderem ein Gaußförmiger Ionenstrahl implementiert, d.h. die Strahlstromdichte ist eine Gaußfunktion der Ortskoodinate x.

$$F_{\mathrm{beam}}(x) = \frac{I/e}{\sqrt{2\pi}\sigma W_{\mathrm{z}}} \exp\left(-\frac{(x-x_{\mathrm{c}})^2}{2\sigma^2}\right)$$

I ist der Strahlstrom, W_{z} die Breite in z-Richtung, σ die Standardabweichung der Gaußfunktion und x_{c} der Ort des Maximums. In Fokussierten-Ionenstrahlanlagen wird der Strahl meist „digital" gescannt, d.h. der Strahl verweilt eine bestimmte Zeit (die Dwell-Time t_{d}) auf einer Stelle x_{c}, wird dann auf eine andere Stelle gerichtet, wo er wieder die Dwell-Time verweilt u.s.w. Man unterscheidet drei Scanmoden:



1. Rastermodus: Es wird in x-Richtung (von links nach rechts) $n_{\mathrm{pixels}} - 1$ mal um ein „Pixel-Spacing" weitergegangen, dann wird um ein Pixel-Spacing in z-Richtung verschoben und die zweite Zeile wieder von links nach rechts gescannt u.s.w. (Bild oben, links).

2. Serpentinenmodus: Wie oben, jedoch wird jede zweite Zeile von rechts nach links durchlaufen (Bild oben, rechts).

3. Stream-Modus: Position und Dwell-Time der Pixel können einzeln definiert werden, sie werden in einem File abgelegt.

Da wir die Steps in die z-Richtung nicht simulieren, nehmen wir an, dass der Strahl in z-Richtung eine Ausdehnung W_{z} hat.

Aufgabenstellung: Erweitern Sie das `beam`-Modul um die Möglichkeit, den Strahl wie oben beschrieben in 3 Moden „digital" zu scannen, wobei wir Schritte in z-Richtung außer Acht lassen. Beim Stream-Modus werden Pixelposition x_{c} und Dwell-Time t_{d} aus einem File gelesen, das x_c und t_{d} in zwei Spalten enthält.

Führen Sie die folgenden Parameter ein:

`[Beam] SCAN_TYPE`: Scanmodus (`'none'`, `'raster'`, `'serpentine'` oder `'stream file'`; `'none'` bedeutet, dass sich die Strahlposition nicht ändert). Default: `'none'`

`[Beam] DWELL_TIME`: Verweilzeit des Strahls auf einem Pixel. Default: 1 ms.

`[Beam] PIXEL_SPACING`: Abstand aufeinanderfolgender Pixel. Default: `0`.

`[Beam] N_PIXELS`: Anzahl der Pixel in einem Scan. Default: `1`.

`[Beam] N_SCANS`: Anzahl der Scans. Default: `1`.

`[Beam] STREAM_FILE`: Name des Files, das die Pixelinformation enthält. Default: ''.

Außer im `'stream file'` Modus sei die Position des ersten Pixels durch den schon in AUFGABE 8 eingeführten Parameter `[Beam] BEAM_CENTER` gegeben. Von dort ausgehend werde der erste Scan in Richtung positiver x-Achse ausgeführt.

Führen Sie Simulationen durch, die Gräben erzeugen, die etwa 1 µm breit und 1 µm tief sind, wobei Sie 1 nA Strahlstrom, ein Pixel-Spacing von 100 nm und ein Gaußsches Strahlprofile mit 100 nm FWHM verwenden. Vergleichen Sie das Ergebnis, wenn Sie die Tiefe in einem Scan erreichen (mit größerer Dwell-Time) mit jenem, wenn Sie 100 Scans mit etwa 100-fach kleinerer Dwell-Time verwenden. Für letztere vergleichen Sie Raster- und Serpentinenscan. Schalten Sie Redeposition ein (`[Setup] REDEP=True`). Schreiben Sie (optional, d.h. wenn Sie noch Lust haben) mit Hilfe eines Skripts ein Stream-File, das in einem Scan einen möglichst horizontalen Boden liefert.

Richten Sie schließlich einen Test für den Raster-Scan ein, in dem Sie mit der kleinen Dwell-Time einen Zeitschritt pro Pixel machen.

Hinweis: Es bietet sich für die Erzeugung der Pixels (Position und Verweilzeit) die Implementierung mit Hilfe eines Generators (`yield` statement) an.

Arbeitsverzeichnis: *miniTopSim/work/Aufgabe11_scan*

Abzugebende Files:

advance.py: `get_velocities()`-Funktion, die die aktuelle Strahlflussdichte ermittelt.

beam.py: Modifiziertes `beam`-Modul.

parameters.db: Ergänzte Parameter-Datenbank.

single_scan.cfg: *cfg*-File, mit dem ein etwa 1 µm breiter und 1 µm tiefer Graben mit einem Scan erzeugt wird.

raster_scan.cfg: *cfg*-File, mit dem ein etwa 1 µm breiter und 1 µm tiefer Graben im Rasterscan-Modus mit 100 Scans erzeugt wird.

serpentine_scan.cfg: *cfg*-File, mit dem ein etwa 1 µm breiter und 1 µm tiefer Graben im Serpentinenscan-Modus mit 100 Scans erzeugt wird.

**.png*: Die zugehörigen Bilder, die die Oberflächen am Ende der Simulation darstellen.

raster_scan_1.cfg: *cfg*-File für den Test (ein Zeitschritt pro Pixel).

test_raster_scan.py: Test.

AUFGABE 12: Regressionstests

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

Background: Automatisiertes Testen erfordert die Bestimmung von Zahlenwerten, deren Richtigkeit durch Vergleich mit einem bekannten Wert bzw. dem Wert eines früheren Laufs festgestellt werden muss. Einfache Beispiele dafür finden Sie im *miniTopSim/work/templates* Verzeichnis, Beispiele für Unittests wurden in AUFGABE 6 behandelt. Regressionstests vergleichen den Output von Simulationen nach Code-Modifikationen mit dem früherer Simulationen bei gleichem Input. Dabei ergibt sich das Problem, dass sich aus numerischen Gründen oder geänderter Implementierung nicht exakt idente Ergebnisse einstellen können, ohne dass der Code fehlerhaft ist. Bei miniTopSim könnte ein adaptives Gitter, das in einer anderen Aufgabe eingebaut wird, zu unterschiedlicher Anzahl und/oder Position der Punkte führen.

Tritt ein Fehler auf, so kann der graphische Vergleich der Oberflächen aus zwei Läufen erste Aufschlüsse geben, siehe AUFGABE 9. Für automatisiertes Testen muss jedoch der Unterschied zwischen zwei Oberflächen quantifiziert werden.

Aufgabenstellung: Schreiben Sie eine Methode `distance()` in der `Surface`-Klasse, die den Abstand zweier Oberflächen berechnet.

- *Der Abstand zweier Oberflächen* sei definiert als Mittelwert des Abstandes der ersten Oberfläche von der zweiten und des Abstandes der zweiten Oberfläche von der ersten.
- *Der Abstand einer Oberfläche von einer anderen* sei definiert als der Mittelwert der Abstände der Punkte der einen Oberfläche von der anderen Oberfläche.
- *Der Abstand eines Punktes von einer Oberfläche* sei definiert als der Minimalwert der Abstände des Punktes von den Segmenten der Oberfläche.
- *Der Abstand eines Punktes von einem Segment* ist der Normalenabstand, wenn der Fußpunkt innerhalb des Segments liegt, andernfalls der Abstand vom nähergelegenen Endpunkt.

Da der Aufwand von der Ordnung $\mathcal{O}(n^2)$ ist, versuchen Sie, ihn mit Whole-Array-Operationen zu implementieren.

Hinweis: Um alle Kombinationen von Punkten einer Oberfläche (self) und Punkten der anderen Oberfläche (other) zu erhalten, konstruieren Sie 2D Arrays mit Hilfe von `xj, xi = np.meshgrid(other.x, self.x)` etc. (siehe auch AUFGABE 5). Beachten Sie, dass sich die Segmente der anderen Oberfläche zwischen (`xj1=xj[:,:-1]`, `yj1=yj[:,:-1]`) und (`xj2=xj[:,1:]`, `yj2=yj[:,1:]`) befinden.

Wiederholen Sie die Simulation aus AUFGABE 1 (etch.cfg) und führen Sie eine ähnliche Simulation mit DELTA_X=0.125 (*etch_dx0_125.cfg*) aus. Bestimmen Sie den Abstand der erzeugten Oberflächen *etch.srf* und *etch_dx0_125.srf*. Schreiben Sie zwei Pytest-Tests, die die Simulationen automatisch ausführen und mit dem halben zuvor bestimmten Abstand vergleichen. Im ersten Fall wird *etch.srf* mit sich selbst verglichen (dieser Test muss erfolgreich sein), im zweiten Fall wird *etch.srf* mit *etch_dx0_125.srf* verglichen (dieser muss fehlschlagen). Markieren Sie den zweiten Test als „expected to fail".

Kopieren Sie *etch.srf* auf *etch.srf_save*. Schreiben Sie einen Test, der *etch.cfg* laufen lässt und die letzte Oberfläche dieser Simulation mit der letzten Oberfläche aus dem *srf_save*-File

vergleicht. Wählen Sie das Kriterium für den Erfolg des Tests wie im vorigen Punkt. Markieren Sie den Test als Regressionstest, sodass er mit *pytest -m regression* läuft. Schreiben Sie schließlich Regressionstests für alle anderen bisherigen Aufgaben (AUFGABEN 4, 7, 8 ,9) und überprüfen Sie, ob sie alle erfolgreich sind.

Arbeitsverzeichnis: *miniTopSim/work/Aufgabe12_regression*

Abzugebende Files:

surface.py: Die um die `distance()` Methode erweiterte `Surface` Klasse.

etch_dx0_125.cfg: *cfg*-File für die Simulation mit `DELTA_X=0.125` ohne automatisches Plotten der Oberfläche.

etch.cfg: *cfg*-File für die Simulation mit `DELTA_X=1` ohne automatisches Plotten der Oberfläche.

dist_etch_1_0_125.py: berechnet die Distanz zwischen den letzten Oberflächen aus *etch.srf* und *etch_dx0_125.srf* (Punkt 2).

test_etch_1_vs_0_125.py: Test, welcher die letzte Oberfläche von *etch.srf* mit sich selbst und mit der letzten Oberfläche von *etch_dx0.125.srf_save* vergleicht.

test_etch.py: Regressionstest, welcher die letzten Oberflächen von *etch.srf* und *etch.srf_save* vergleicht. Achten Sie darauf, dass der Test mit Pytest auch vom miniTopSim-Verzeichnis aus gestartet werden kann.

AUFGABE 13: GUI — Eine Section

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

Aufgabenstellung: Erstellen Sie ein graphisches User-Interface mit Hilfe von Tkinter unter Verwendung von themed widgets (`tkinter.ttk`), welches eine Section der Parameter-Datenbank *parameters.db* einliest. Der Name der Section soll als Commandline-Argument beim Aufruf angegeben werden:

python3 <path-to-minitopsim>/gui.py <section-name>

Die in der Parameter-Datenbank definierten Defaultwerte sollen angezeigt werden und sich ändern lassen. Flags sollen mittels Checkboxen und numerische bzw. String-Werte in Entries (`Entry`) oder Spinboxes (`Spinbox`) dargestellt werden. Verwenden Sie ein Grid Layout.

Wird ein Parameter verändert, soll die Bedingung aus der Datenbank überprüft werden und bei Verletzung die Änderung verworfen und eine Fehlermeldung ausgegeben werden.

Weiters sind OK und Cancel Buttons vorzusehen, um das GUI zu beenden und im Fall von OK die Parameter auf ein cfg-File mit festem Namen (*beispiel.cfg*) zu schreiben.

Arbeitsverzeichnis: *miniTopSim/work/Aufgabe13_gui*

Abzugebende Files:

gui.py: GUI Script zum Erstellen eines cfg-Files. Aufruf wie oben angegeben.

beispiel.cfg: Ein erzeugtes cfg-File.

Hinweis: Ein gutes Tkinter Tutorial ist unter https://tkdocs.com/tutorial/ zu finden (wählen Sie „Python" aus). Weitere nützliche Quelle ist die Dokumentation der Python-Standardbibliothek (https://docs.python.org/3/library/tk.html) sowie das Buch A.D. Moore: Python GUI Programming with Tkinter, Packt, 2018 (online zugänglich an TU Wien).

AUFGABE 14: GUI — Alle Sections

Bitte lesen Sie das `README.md` File des miniTopSim-Projekts!

Erweitern Sie das graphische User-Interface aus AUFGABE 13, sodass alle Sections bearbeitet werden können. Dabei soll jede Section in einem Tab angezeigt werden. Die Angabe der Section auf der Commandline wird dadurch überflüssig und soll daher beseitigt werden.

In einem weiteren Schritt sehen Sie vor, dass die Default-Parameter mit den Werten eines bestehenden cfg-Files überschrieben werden können, welches als Commandline-Argument übergeben wird. D.h. der Aufruf soll nun über

python3 <path-to-minitopsim>/gui.py <cfg-file>

erfolgen, wobei die Angabe des cfg-Files optional ist. Vor dem Lesen dieses Files soll weiterhin *parameters.db* gelesen werden.

Weiters soll beim Verlassen des GUI nach dem Filenamen des cfg-Files gefragt werden, in das die Parameter gespeichert werden sollen.

Optional: Implementieren Sie zu jedem Parameter einen Tooltip, der die Erklärung des Parameters aus der Datenbank anzeigt.

Arbeitsverzeichnis:

miniTopSim/work/Aufgabe14_gui

Abzugebende Files:

gui.py: GUI Script zum Erstellen eines cfg-Files. Aufruf wie oben angegeben.

beispiel.cfg: Das cfg-File aus AUFGABE 13.

beispiel2.cfg: Das erzeugte cfg-File.

beispiel2.png: Ergebnis-Plotfile eines miniTopSim-Runs mit dem cfg-File *beispiel2.cfg*.