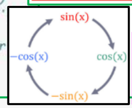
Aufgabe 1:
Kategorisieren von Modellen: **1.diskret:** kleine abzählbare Schritte vs. **2. kontinuirlich:** stetig **3. deterministisch:** Anfangsbedingungen, keine Zufallswerte vs. **4.stochastisch:** Zufallsprozesse und Wahrscheinlichkeiten **Bsp: Räuber-Beute:** kontinuerlich, deterministisch **Was Diskretisierung?** Kontinuierlicher Verlauf in Diskrete Zeitschritte(Robot, Quarter Car, Water waves). **Diskretes Modell** endlich viele Zustände zb TSP **Was Simulated Anneiling anders?** Tsp diskretes Simulationsproblem, Simmulated Anneiling stochastisch **Simulationspipeline:** **1.Modellierung:** beschreiben was simuliert (zB Navier-Stokes) **2.Aufbereitung:** Modell zur Behandlung auf Rechner aufbereitet (Diskretisierung, Algorithmus, technische Zeichnungen) **3. Implementierung:** Performante Implementation Zielsystem (Programmierung Parallelrechnersysteme) **4.Visualisierung:** der Ergebnisse, **5. Bewertung:** (Validierung) wie verlässlich Ergebnisse **6. Einbettung:** Integration Simulation in Kontext (zB Simulation CAD Designer zur Verfügung stellen) **Unterschied Modellierung und Aufbereitung:** Modelierung beschrieben, was simuliert werden soll. Aufarbeitung zu simulierende Sachverhalt genauer beschrieben, zB technische Zeichnung
Sinn und Zweck Simulation: **1.Ursachenforschung** (Geburtenrate,Erdbeben), **2.Vorhersage** (Prognose Population), **Optimierung** (optimale Jagdquote, Fahrdynamik , Schadstoffausstoß)
Warum Simulation besser? Experimente **1.schwieriger, 2. Ethisch nicht vertretbar, 3. Teurer und zeitaufwändiger, 4. Gefährden mensch und umwelt** **Anwendungsgebiete:** Natur-, Ingenieur-, Wirtschaftswissenschaften **3 Säulen (Simulation, Theorie, Experimente) Wechselwirkung:** Sim.-> Theo.: Anpassung theoretischer Modelle durch Simulationserg. **Warum Simulation 3. Säule?** Ergänzt Exp., Theo., günsitger, vorhersage, optimierung, sicher… **Validierung und Verifikation:** **Verifikation:** korrekte Implementation sicherstellen, Modell richtig gebaut?, überprüfen Code auf Fehler, Logik Fehler, Verwendung Testfälle zum überprüfen **Validierung:** Spiegelt Modell Realität wieder? Richtiges Modell gebaut? (Realitätstest, Plasibilitätskontrolle, Vergleich mit bekannten Ergebnissen)
Aufgabe 3 Robot: **Direkte Kinematik:** Position des Roboters an Stellgrößen berechnet
Inverse Kinematik: Stellgrößen anhand Position des Roboters berechnet
Verfahren, das Bahnkurve für Bewegung Gegenstand über beliebig geformtes Hindernis automatisch generiert: Aüßere Begrenzungslinien den Hindernissen entsprechend der Höhe, Breite Gegenstandes nach link nach oben und nach rechts verschieben. Evtl. Offsetkurven durch weitere Kurvenstücke verbinden .
Verifikation: **Probleme** Berechnung inverse Kinematik bei Newton Verfahren: 1. **Jacobi Matrix ist Singular**: Lösung nicht immer logisch. 2. **Schlechte Startwerte**, dann Konvergiert nicht, also keine Lösung. **Lösung:** letzte Newton Iteration als Startwert (oder Überprüfung nach festgelegter Anzahl Iteration nicht konvergiert, Fehler ausgeben) Verlässichkeit Näherungslösung muss bestimmt werden **Validation:** Simulation für Einsatzzweck geeignet? Vergleich mit Realität würden wir Unterschiede feststellen, da wir nur die Bewegungssequenz betrachten und physikalische Größen wie Masse, Kräfte und Reibung vernachlässigt haben, zB Schwingungsbewegungen transportierter Gegenstand (auch wurde nicht definiert ob der Greifarm selbst kollidiert, sondern nur das Objekt)
TSP: Brute Force Algorithmus: geht alle n! Permutationen Städte n! durch **Vorteil:** einziger Algorithmus der optimale Route garantiert findet **Nachteil:** hoher Rechenaufwand, lange Laufzeit schon bei kleiner Anzahl Städte -> maximale Rechenleistung nur erzielt, wenn Algo. Parallelisiert weden kann
-> wenn n! gesucht, testen in TR **Greedy-Alg**: eilt zur nächstgelegenen Stadt (kürzeste Strecke in Distanzmatrix), um die kürzeste Route zu finden, indem er schrittweise lokale Minima bestimmt. Dies führt oft nicht zur optimalen Lösung des Problems. **2-Opt-Algorithmus: Kontinuierliche Verbesserung:** Beliebige Route begonnen und schrittweise verkürzt, bis keine Verbesserung möglich ist. Ergebnis ist eine Folge von Routen mit ständig abnehmender Länge. **Kantenvertauschung:** Routen mit sich kreuzenden Kanten nicht optimal. 2-Opt verbessert die Route durch Austausch von zwei Kanten. Er sucht nicht explizit nach Kreuzungen, reduziert diese aber indirekt, da kürzere Routen weniger Kreuzungen haben. **Menge an Edge Swaps:** **First Improvement:** Zwei Kanten der Ausgangsroute getauscht, bis die erste Route gefunden wird, die kürzer als die Ausgangsroute ist
Steepest Descent: Alle Möglichkeiten, zwei Kanten der Ausgangsroute zu tauschen, untersucht und unter allen diesen Möglichkeiten die kürzeste Route gewählt.
Steepest Descent vs First Improvement Unterschied: bei beiden Algorithmen betrachtet man eine Ausgangsroute. **SD:** alle Möglichkeiten, zwei Kanten der Ausgangsroute zu tauschen, untersucht und unter allen diesen Möglichkeiten die kürzeste Route ausgewählt **FI:** solange zwei Kanten Ausgangsroute getauscht, bis die erste Route gefunden wird, die kürzer als die Ausgansroute ist **Simmulated Annealing**: Überwindung lokale Minima, um globale Minima zu bekommen durch Temperaturabnahme. **Ausgangspunkt:** Beliebige Route **Alternative Route:** Zufällig bestimmt durch Edge Swapes. **Vergleich:** Länge der ursprünglichen Route mit der alternativen Route vergleichen. **Entscheidungsprozess:** Eine Pseudeo-Zufallszahl z(0-1) bestimmt, ob die alternative Route angenommen wird(z<=p). **Lösungsverfahren für Optimierungsprobleme die auf Zufallsprinzip beruhen. Was beachten wenn man Lösungsverfahren in Softwareprodukt integriert?** Zufallsprinzip sollte mit Pseudozufallszahlen realisiert werden, die es ermöglichen reproduzierbare Ergebnisse zu liefern(festgelegter Random seed).
Quartercar:Verifikation: Bei numerischen Simulationsmethoden stellt sich Frage nach der Zuverlässigkeit der numerischen Näherungswerte; Je kleiner h: hohe Genauigkeit, lange Rechenzeit, Rundungsfehler -> Bei Verifikation überprüfen ob Schrittweite h richtig gewählt wurde. **Wie überprüfen ob Schrittweite richtig gewählt wurde?** Man berechnet einfaches Fahrbahnmodell analytisch und vergleicht numerische Lösung dann mit der exakten analytischen Lösung. Alternativ kann man auch Lösung mit unterschiedlichen Schrittweiten und unterschiedlichen numerischen Verfahren berechnen und Lösungen miteinander vergleichen. **Validierung:** brauchbare Ergebnisse für die Fahrzeugdynamik? Quartercar gibt nur grobe Ergebnisse; Steifigkeit des gesamten Fahrzeugs oder die Massenverteilung auf die einzelnen Räder nicht berücksichtigt ->überprüfen ob sich Feder-Masse-Schwinger in echt so verhält. **Wofür Simulation gut praktische Anwendung?** Untersuchung Fahrzeugverhalten bei plötzlichen Hindernissen auf unebener Straße -> Verbesserung Sicherheitsfaktor, bessere Kontrolle/Stabilität. Analyse Stoßdämpfer, Untersuchung Dämpfungscharakteristiken/ Leistungsfähigkeit -> Auswahl/ Anpassung von Stoßdämpfern um optimale Dämpfungseigenschaften zu gewährleisten **Woraus besteht Modell?** Feder, Dämpfer, Masse, Straßenprofil
Unterschied Euler, Lax-Wendroff, Runge Kutta: Euler: +einfache Implementation, +geringer Rechenaufwand pro Schritt, -weniger genau bei größeren Schrittweiten, -kann instabil sein für steife Gleichungen **Lax-Wendroff:** **Half step schritte** (2. Ordnung Methode) -> berechnung zwischenschritte verbessert Genauigkeit, durch Halbschritte Berechnung an Gitterpunkten genauer, **Bedeutung Indizes fk+1/2,i+1/2:** Beschreiben Größe f zu bestimmten Zeitpunkt und Ort. I+1/2: Ort zwischen Gitterpunkten i und i+1, k+1/2 Zeitpunkt zwischen k und k+1, für waterwaves (Strömungsmechanik) verwendet, +bessere Genauigkeit für hyperbolische Gleichungen, +kann Schockwellen und diskontinuierliche Lösungen besser handhaben, -komplexer zu implementieren, -höherer Rechenaufwand pro Schritt, -weniger geeignet für parabolische, elliptische Gleichungen. **Anfangsbedingungen implementieren:** Wasserberg anheben indem man anfängliche Höhe h(x,0) in bestimmten Bereich erhöht, Anfangsgeschwindigkeit, indem Anfangswerte für Geschwindigkeitsfelder u(x,0= und v(x,0) festlegt **Runge-Kutta:** **adaptive Schrittweitensteuerung:** anpassung der Schrittweite h während numerischen Integration dynamisch an Problem, klassisch 4 Stufen, **Ziel:** Optimierung Genauigkeit und Effizienz, + kleine Schrittweiten in kritischem Bereich, +reduziert Rechenzeit, da unnötige Berechnungen vermieden werden. +sehr hohe Genauigkeit, kann größere Schrittweiten verwenden, wodurch weniger Schritte nötig sind, -hoher Rechenaufwand pro Schritt, -komplex zu implementieren
Waterwaves: **Wofür kann Simulation gut sein?** Wasserwellen für Surfer, Tsunami Schäden abschätzen und vermeiden. **Ausgangssituation initialisieren:** höhe Wasser wird an allen Gitterpunkten auf konstanten Wert gesetzt. An Gitterpunkt wird Wert entweder erhöht oder erniedrigt. Geschwindigkeiten werden mit 0 initialisiert **Visualisierung:** Farbskala für Geschwindigkeit, niedrigere Dunkelblau und hohe hellblau bis weiß. **Validierung:** wie realistischer machen? **Problem:** Bewegung Wellen flacht im laufe der Zeit nicht ab ->liegt an Navier Stokes gleichungen, die auf Erhaltung von Masse und Impuls beruhen, Dämpfungsmechanismen wie Reibung werden nicht berücksichtigt. **Lösung:** man reduziert nach jedem Schritt die Geschwindigkeit des Wassers. **Zu Differenzieren:** Zentral am genauesten, da Mittelwert der Änderung in beide Richtungen berücksichtigtm somit Fehler der Approximation minimiert **Räuber Beute:** beschrieben durch gewöhnliche DGL, lassen sich nicht analytisch lösen, da nichtlineare Wechselwirkung, es wird Runge Kutta für Approximation verwendet.Populationsentwicklung Räuber-Beute mit Dynamik und Wechselwirkung **Zweck:** Ursache: Populationsschwankung, Epidemiologie Viren auf Wirte, Vorhersage: Wildtiermanagement Vorhersage zum planen, Landwirtschaft Auswirkung Schädlinge auf Nutzpflanzen, Optimierung: Eingriffe Jäger zum Schutz, Fischen Raubfische, Pestizidmanagement **Moddelierung:** Vernachlässigt Klima, Mensch etc., Annahme Beutetiere unbegrenzt Nahrung, Vermehrung durch Räuber beeinflusst (nicht natürlicher Tod), bestand Population über Zeit anzeigen, Beutetiere vermehren sich und werden gefressen, Räuber vermehren sich nur bei ausreichend Nahrung Beutetiere und sterben mit bestimmmter Rate

```python
#Implementation für einfache Gleichung
x = 0.5 #Startwert
for k in range(10): #10 Iterationen
    f = np.cos(x)-x #f(x)=cos(x)-x=0
    fp = -np.sin(x)-1 #1.Ableitung f'(x)
    dx = -f/fp #Aus Formel -f(x)/f'(x)
    x = x + dx #Aus Formel x=x-f(x)/f'
    print('x = ',x)
    #Abbruch wenn Differenz dx kleiner
    #Maschinengenauigkeit ist
    if abs(dx) < np.finfo(float).eps:
        break
```

einzelne Gleichung:

$$x_{k+1} = \hat{x}_k - \frac{f(\hat{x}_k)}{f'(\hat{x}_k)}$$



$$n \geq \frac{\ln\left(\frac{T_{low}}{T_{high}}\right)}{\ln(\beta)}$$

$$C(n,2) = \frac{n(n-1)}{2}$$

$$p = \begin{cases} e^{\frac{l_0 - l_1}{T}} \\ 1 \end{cases}$$

normaler PC: $100\,GFLOPS = 10^{11} FLOPS$
Supercomputer FRONTIER: $1685 PFLOPS \approx 1{,}6 \cdot 10^{18} FLOPS$
$A = n!$ $\quad T_n = \frac{A_n}{FLOPS}$ in Sekunden

```python
#Newton Methode für System
# sin(x)-y=0
# x^2+y^4-1=0
x = np.array([0.0,1.0])
for k in range(10):
    f=np.array([np.sin(x[0])-x[1],x[0]**2+x[1]**4-1])
    J=np.array([[np.cos(x[0]),-1.0],[2*x[0],4*x[1]**3]])
    dx=np.linalg.solve(J,-f)
    x = x + dx
    print('x = ',x)
    if max(abs(dx)) < np.finfo(float).eps:
        break
```

$$J = \begin{pmatrix} f_{1x_1}, f_{1x_2} \\ f_{2x_1}, f_{2x_2} \end{pmatrix}$$

$$Euler: x^{(k+1)} = x^{(k)} + h\dot{x}^{(k)}$$

$z_{k+1} = z_k + h \cdot F(t_k, z_k)$ in Vektorschreibweise:

$$\rightarrow \begin{pmatrix} z0_{k+1} \\ z1_{k+1} \end{pmatrix} = \begin{pmatrix} z0_k \\ z1_k \end{pmatrix} + h \cdot \begin{pmatrix} f_0(t_k, z0_k, z1_k) \\ f_1(t_k, z0_k, z1_k) \end{pmatrix}$$

```python
def robotInverseKinematics(T,alpha=np.pi/2.0,beta=np.pi/2.0):
    h, w, d, l1, l2, l3, r = robotValues()
    #Definiere die Funktionen f1 u. f2
    def f1(alpha, beta):
        T_x = l1 * np.cos(alpha) - l2 * np.cos(alpha+beta)
        return T[0] - T_x
    def f2(alpha, beta):
        T_y = l1*np.sin(alpha)-l2*np.sin(alpha+beta)-l3-2*r
        return T[1] - T_y
    #Definiere Jakobian-Matrix
    def jacobian(alpha, beta):
        J = np.zeros((2, 2))
        J[0, 0] = l1*np.sin(alpha)-l2*np.sin(alpha+beta)
        J[0, 1] = -l2*np.sin(alpha+beta) #oben rechts
        J[1, 0] = -l1*np.cos(alpha)+l2*np.cos(alpha+beta)
        J[1, 1] = l2*np.cos(alpha+beta)
        return J
    #Newtons-Verfahen iteration
    max_iterations = 100
    tolerance = 1e-6
    iteration = 0
    #Newton Verfahren durchführen
    for k in range(max_iterations):
        #Berechne Funktion mit dem aktuellen Winkel
        f = np.array([f1(alpha, beta), f2(alpha, beta)])
        #überprüfen ob berechnete Funktionswerte f nah genug
        if np.max(np.abs(f))<tolerance:
        #Berechne Jacobi Matrix mit aktuellen Winkeln
        J = jacobian(alpha, beta)
        #1. Fehlererkennung: Jacobi Matrix singulär?
        try:
            delta_theta = np.linalg.solve(J,-f)
        except np.linalg.LinAlgError:
            return None, None, iteration, "Jacobian singular"
        #Update die Winkeln
        alpha += delta_theta[0]
        beta += delta_theta[1]
        #Überprüfen ob Änderung Winkel alpha, beta sehr klein
        #Hinweis dass sich Lösung stabilisiert
        if max(abs(delta_theta)) < 0.0001:
            break
        iteration+=1
    #2. Fehlererkennung, konvergiert nicht
    if iteration == max_iterations:
        return None, None, iteration, "nicht konvergiert"
    return alpha, beta
```

```python
import numpy as np
def euler(t0,tn,h,x0):
    n = int((tn-t0)/h)
    t = np.linspace(t0,tn,n+1)
    x = np.zeros(n+1)
    x[0] = x0
    #Euler Verfahren Schritte ausführen
    for k in range(n):
        x[k+1] = x[k]+h*f(t[k],x[k])
    return t, x
```

```python
def f(t,x):
    f = -6*t-x/(1+t)
    return f
```

```python
def f2(t,z):
    f2 = np.array([z[1],np.cos(t)-z[1]-np.sin(z[0])])
    return f2
```

```python
import numpy as np
def euler2(t0,tn,h,z0):
    n = int((tn-t0)/h)
    t = np.linspace(t0,tn,n+1)
    z = np.zeros([len(z0),n+1])
    z[:,0] = z0
    for k in range(n):
        z[:,k+1] = z[:,k]+h*f2(t[k],z[:,k])
    return t, z
```

Stellen Sie das Anfangswertproblem

$$\ddot{x} + \sin(x) = t, \quad x(0) = 1, \dot{x}(0) = 0,$$

mithilfe von Zustandsvariablen dar.

Zustandsvariablen $z_0 = x$, $z_1 = \dot{x}$:

$$\dot{z}_0 = z_1$$
$$\dot{z}_1 = t - \sin(z_0)$$

```python
def quarterCarEuler(h=0.01,t0=0.0,t1=1.0):
    n = int((t1-t0)/h) #Anzahl der notwendigen Schritte
    t = np.linspace(t0,t1,n) #Zeitvektor von t0 bis t1
    z = np.zeros([2,n]) #Array mit der Länge n,mit 0en aufgefüllt
    z[:,0] = np.array([y_0,0.0]) #Anfangsbedingungen
    #Euler Verfahren              #position y_0 und Geschw. 0.0
    for k in range(n-1):
        z[:,k+1] = z[:,k] + h * quarterCarOde(t[k], z[:,k])
        #Berechne neuen Zustand durch hinzufügen Produkt aus
        #Schrittweite h und Ableitung (quartercarOde)
    y = z[0,:] #extrahiere Position (erste Zeile Zustandsmatrix)
    return t, y
```

```python
def quarterCarOde(t,z):
    z0=z[0] #Substitution z0=y, z1=y'
    z1=z[1]
    x = v*t #position auf x-Achse muss berechnet werden, da nicht explizit übergeben
    f, fp = quarterCarRoad(x) #Werte f, fp für die Gleichung z1p notwendig
    # wichtig nicht f(x*v) da es in einem numpy.float64 error resultiert,
    #f ist hier eine Variable und nicht als Funktion definiert,
    #weshalb sie auch nicht als Funktion aufgerufen werden kann
    z1p = (-c*(z0-y_0-f)-k*(z1-v*fp))/m
    zp = np.array([z1,z1p]) #ersten Ableitungen übergeben z0'=z1=z[1], z1'=z1p
    return zp
```

$$h(x, y, t) = \frac{e^{-x^2-y^2}}{1+10t}, \quad (x, y) \in [-2, 2]^2, \quad t \in [0, 1]$$

$$\begin{pmatrix} u(x, y, t) \\ v(x, y, t) \end{pmatrix} = (1-t)\begin{pmatrix} x \\ y \end{pmatrix} + t\begin{pmatrix} -y \\ x \end{pmatrix}, \quad (x, y) \in [-2, 2]^2, \quad t \in [0, 1]$$

```python
def tspSimulatedAnnealing(dist, route=[], beta=0.99, T_high=100, T_low=10):
    #Anzahl der Städte
    n = dist.shape[0]
    #überprüfen ob Route None oder Empty ist
    if route is None or len(route) ==0:
        route = np.arange(n) #initialisiere Route als Sequenz der Städte
    #Liste zur Speicherung aller ROuten erstellen und Startroute hinzufügen
    routes = [route] #erstellt leere Liste, fügt Startroute als Element hinzu
    #Initialisieren der Temperatur
    T = T_high
    #falls man random seed festlegen soll random=4777 in random.seed(seed)
    #Simulated Anneiling durchführen
    while T > T_low:
        #zufällige Indizes i und j generieren
        i = random.randint(0, n-1)
        j = random.randint(0, n-1)
        #edge swaps durchführen
        route_swap = tspSwapEdge(route, i, j)
        #längen der aktuellen und der geswapten route vergleichen
        len_route = tspLength(dist, route)
        len_route_swap = tspLength(dist, route_swap)
        #Akzeptieren der swapped_route wenn sie kürzer ist
        if len_route_swap < len_route:
            route = route_swap
        #akzeptiere Route mit bestimmter Wahrscheinlichkeit basierend auf Temp
        else:
            #p Akzeptanz schlechte Lösung wird immer kleiner
            p = np.exp(-(len_route_swap-len_route)/T)
            z = random.random()#zufälliger Schwellenwert z der Akzeptanz beeinflusst
            if z <= p:#p muss höher sein als zufälliger Schw. z um schlechtere Lös.zu akz.
                route = route_swap
        #Füge aktuelle Route zur Liste der Route
        routes.append(route)
        #aktualisieren der Temperatur
        T *= beta
    return route, routes
```

```python
def tspSwapEdge(route,i,j):
    routeSwap = route.copy()
    if j < i:
        i, j = j, i
    for k in range(j-i+1):
        routeSwap[i+k] = route[j-k]
    return routeSwap
```

```python
def tspLength(dist, route):
    length = 0
    for i in range(len(route) - 1):
        city1 = route[i]
        city2 = route[i + 1]
        length += dist[city1][city2]
    length += dist[route[-1]][route[0]]
    return length
```

```python
import numpy as np
import matplotlib.pyplot as plt
def velocityFieldPlot(t=0.5):
#alten Plot löschen und neuen erstellen
#fig = plt.gcf()
#ax = fig.gca()
#ax.clear()
#Gitterpunkte erzeugen
#20 Punkte im Bereich -2,2 für x,y
x = np.linspace(-2, 2, 20)
y = np.linspace(-2, 2, 20)
#Erstellt Gitter von Punkten
#im Bereich[-2, 2]x[-2, 2]
X, Y = np.meshgrid(x, y)
#wenn keine Funktion muss zb t=0.5 defin
# Berechnet die Komponenten des
#Geschwindigkeitsvektors für jeden Punkt
u = (1 - t) * X + t * (-Y)
v = (1 - t) * Y + t * X
#Erstellen der figur und Achsen
fig,ax = plt.subplots(figsize=(5,5))
#zeichnen Richtungsfeld mit quiver()
field = ax.quiver(X, Y, u, v)
return field
```

Vorwärtsdifferenzieren: $\frac{\partial f(x,t)}{\partial x}$ *mit* Δx: $\frac{f(x+x\Delta,t)-f(x,t)}{\Delta x}$

Rückwärtsdifferenzieren: $\frac{\partial f(x,t)}{\partial x}$ *mit* Δx: $\frac{f(x,t)-f(x-\Delta x,t)}{\Delta x}$

Zentrales Differenzieren: $\frac{\partial f(x,t)}{\partial x}$ *mit* Δx: $\frac{f(x+\Delta x,t)-f(x-\Delta x,t)}{2\Delta x}$

```python
import numpy as np
import matplotlib.pyplot as plt
def heightFunctionPlot(t):
    #Gitter erzeugen
    x = np.linspace(-2, 2, 50)
    y = np.linspace(-2, 2, 50)
    X, Y = np.meshgrid(x, y)
    #Berechnen der Höhe h(x,y,t) bas. auf geg. funk.
    h = (np.exp(-X**2 - Y**2)) / (1 + 10 * t)
    #Erstellen figur und 3D-Achsen
    fig = plt.gcf()
    ax = fig.add_subplot(projection='3d')
    ax.clear()
    ax.set_zlim(0,1) #Setze Grenzen der z-Achse
    #Erstelle 3D-Oberflächenplot
    surf = ax.plot_surface(X, Y, h, cmap='viridis')
    return surf,
```