

Hilfsfunktionen für Geometrie in der Ebene.

```
# ACHTUNG: numpy arbeitet mit Winkeln in Radian; Angaben in Grad  
# muß man also umrechnen!  
# Conversion of degree to radian for numpy's trigonometric functions.  
RAD2DEG = 180.0/np.pi  
DEG2RAD = 1/RAD2DEG  
# In our "floating-point-geometry", we cannot handle very small lengths:  
EPSILON = 10 ** (-8)  
  
# Points are numpy-arrays:  
# numpy bietet sehr viel "Lineare-Algebra-Funktionalität" (Vektoren,  
# Matrizen, Addition, Multiplikation, Norm, ...), die wir natürlich  
# hier verwenden wollen.  
# Der Stern vor dem Argument-Namen bedeutet: Diese Funktion übernimmt  
# eine variable ANZAHL von Argumenten (in Form einer Liste). Der Witz  
# der Sache ist: npp(x,y) und npp([x,y]) liefern beide dasselbe  
# Ergebnis.  
def npp(*args):  
    "Convert arguments to np.array"  
    if len(args) == 1:  
        # Only one argument: Should be a list or tuple  
        return np.array(list(args[0]), dtype=float)  
    return np.array(list(args), dtype=float)  
  
# Für Programme, die Punkte in graphische Darstellungen verwandeln,  
# sind Zahlenausgaben in "scientific notation" (also 1.0E-3 statt 0.001)  
# störend, daher vermeiden wir das:  
def chop(num, epsilon = 10**(-5)):  
    "Chop small numbers; to avoid scientific notation"  
    # Nächste ganze Zahl an num:  
    numi = np.round(num)  
    if np.abs(numi-num) < epsilon:  
        return f'{int(numi)}'  
    # Formatierte Ausgabe mit 5 Nachkommastellen:  
    return f'{num:.5f}'  
  
# Ausgabe eines Punktes als String:  
def out(p):  
    "String representation of point p"  
    return f'({chop(p[0])},{chop(p[1])})'  
  
# (Euklidische) Norm (Länge) eines Vektors:  
def get_norm(v):  
    "Compute the norm of v"  
    # Wir verwenden hier die bereits in numpy vorhandene Funktionalität  
    # (get_norm ist also nur ein "wrapper" für np.linalg.norm)  
    return np.linalg.norm(v)  
  
# Bringt einen Vektor (ungleich Nullvektor) auf Länge 1:  
def get_normalized(v_v):  
    "return v_v/norm(v_v)"  
    len_v = get_norm(v_v)  
    # Vermeide Division durch sehr kleine Zahlen:  
    if len_v > 10**(-8):  
        return v_v/len_v  
    # ELSE hier _implizit_, durch return in der if-clause:  
    return None  
  
# Normalvektor:  
# For two-dimensional vectors (nicht Null), we can easily find a  
# (normalized) normal vector:  
def get_normalvector(v_v):  
    "Rotate v_v by +pi/2"  
    # numpy-Funktion flip dreht die Reihenfolge um, und die Multiplikation  
    # v1*v2 erfolgt _komponentenweise_ (nicht verwechseln mit dem inneren  
    # Produkt np.inner(vq, v2)!!!!)  
    return np.flip(v_v)*np.array((-1,1), dtype=float)  
  
# Normalisierter Normalvektor  
def get_normalized_normal(v_v):  
    "Rotate v_v by +pi/2 and normalize"  
    n_v = np.flip(v_v)*np.array((-1,1), dtype=float)  
    return get_normalized(n_v)  
  
# Eine Klasse, die Geraden in der Ebene implementiert: Übernimmt als  
# "initialisierende Argumente" zwei Punkte (die nicht zu nahe sein  
# sollten, um numerische Probleme zu vermeiden).  
class PlaneLine:  
    "Class: Line in the plane, determined by two (distinct) points."  
    def __init__(self, p_p, p_q):  
        if get_norm(p_p-p_q) < EPSILON:
```

```

print('Points too close in PlaneLine.__init__()!!!!')
# Speichere "lokale Kopien" der übergebenen Punkte - würde
# man hier einfach self.p_p = p_p schreiben, würde self.p_p
# nur einen _Verweis_ auf den Punkt p_p enthalten (der später
# geändert werden könnte)
self.p_p = np.copy(p_p)
self.p_q = np.copy(p_q)
# Normalvektorgleichung der Geraden:
self.equation = get_line_equation(p_p,p_q)
# Normalisierter Richtungsvektor:
self.v_v = get_normalized(p_q-p_p)
# Normalisierter Normalvektor:
self.n_v = get_normalized_normal(p_q-p_p)
# Die rechte Seite der Normalvektorgleichung <x,n> = c
self.c_c = np.inner(self.n_v, self.p_p)

# String-representation einer Instanz dieser Klasse:
def __str__(self):
    return f'{out(self.p_p)}-{out(self.p_q)}: {self.equation}.'

# "Aufräum-Funktion", wenn ein Objekt dieser Klasse nicht
# mehr gebraucht wird
def __del__(self):
    pass

# Auswertung der Gleichung in einem Punkt:
def eval_equation(self, p_p):
    "Evaluate line equation for point p_p"
    return np.inner(self.n_v, p_p) - self.c_c

# Fußpunkt des Lots:
def basepoint(self,p_p):
    "Compute base point of orthogonal projection"
    normal_equation = get_line_equation(p_p,p_p+self.n_v)
    return get_solution_xy([self.equation, normal_equation])[0]

# Spiegelung eines Punktes:
def reflect(self,p_p):
    "Return reflection of point p_p at self"
    return 2*self.basepoint(p_p) - p_p

# Parallel durch einen gegebenen Punkt
def parallel(self,p_s):
    "Return parallel line through p_s"
    base_p = self.basepoint(p_s)
    return PlaneLine(
        self.p_p + (p_s - base_p), self.p_q + (p_s - base_p)
    )

# Normale durch einen gegebenen Punkt
def normal(self,p_p):
    "Return orthogonal line through p_p"
    return PlaneLine(p_p, p_p + self.n_v)

# Streckensymmetrale von zwei Punkten:
def plane_line_bisector(p,q):
    """Return line bisector (a.k.a. 'Streckensymmetrale') of
    points p, q."""
    return PlaneLine((p+q)/2, (p+q)/2 + get_normalized_normal(q-p))

```