FACOLTÀ DI INGEGNERIA
CIVILE E INDUSTRIALE

SAPIENZA
UNIVERSITÀ DI ROMA

Elaborato finale per il conseguimento
della Laurea in Ingegneria meccanica

Software for synthesis of function generators cam-follower systems for rapid prototyping

Candidato: Federico Ciotti
Matricola: 1593763

Relatore: prof. Nicola Pio Belfiore
SSD: ING-IND/13

Abstract. The aim of this project is the development of an open source software for the design of cam-follower mechanisms assigned the function of the follower travel or a set of heights to be linked with a given interpolation method. The software is addressed to research and rapid prototyping, thus is provided the option of exporting the result as collection of points in CSV form or directly as tridimensional STL model.

Lo scopo di questo progetto è lo sviluppo di un software open source per la progettazione dimeccanismi a camma assegnata la funzione del movimento traslante della punteria ovvero un insieme di alzate da collegare tramite un'interpolazione assegnata. Il software è indirizzato alla ricerca ed alla prototipazione rapida, pertanto si è prevista la possibilità di esportare il risultato sotto forma di collezione di punti in formato CSV o direttamente come modello tridimensionale in formato STL.

FACOLTÀ DI INGEGNERIA
CIVILE E INDUSTRIALE

SAPIENZA
UNIVERSITÀ DI ROMA

1 Introduction

The program interface is an interactive console written in *Python 3* and it makes use of different libraries:

- *numpy* for array processing
- *matplotlib* for plotting
- *shapely* for geometric analysis
- *scipy* for interpolation
- *sympy* for expression parsing
- *pyclipper* for polygon offsetting
- *stl* for stl manipulation

2 Usage

Command parsing relies on *argparse* module, which takes charge of help generation, so every command's documentation may be requested with `-h` or `--help` option. Avalaible command are listed here:

```
usage: {help,exit,gen,update,load,save,draw,export,sim} ...

positional arguments:
  {help,exit,gen,update,load,save,draw,export,sim}
    help                show this help message
    gen                 generate, unspecified variables set to default
    update              update, unspecified variables unmodified
    load                load from file
    save                save to file
    draw                plot representation
    export              export stl model
    sim                 dynamic simulation
```

`gen` (and `update`), `export` and `sim` are discussed thoroughly in the following sections.

3 Follower travel generation

```
usage: gen travel [-h] [-k {spline,linear,harmonic,cycloidal,parabolic,polynomial}]
                  [--order O] [-n N] [--steps S] [--x0 X0] [--x1 X1]
                  (--input IN | --function F)

optional arguments:
  -h, --help            show this help message and exit
  -k {spline,linear,harmonic,cycloidal,parabolic,polynomial}
                        kind of interpolation (default: linear)
  --order O, -o O       spline/polynomial order (default: 3)
  -n N                  repetitions per cycle (default: 1)
  --steps S, -s S       interpolation steps (default: 10000)
  --x0 X0, -a X0        lower bound of function evaluation (default: 0)
  --x1 X1, -b X1        upper bound of function evaluation (default: 1)
  --input IN, -i IN     input file (default: None)
  --function F, -f F    function of x (default: None)
```

The travel is generated as discretization of a function between given bounds or from interpolation of points from an input file (list of semicolon-separated fraction of unit length and heigth) with a given method.

In the first case the function argument is parsed with *sympify* into a *SymPy* expression; is converted into a lambda function; and it is discretized:

```
f = lambdify(x, sympify(f), modules="numpy")
x = np.linspace(x0, x1, num=steps)
y = f(x)
```

In the second case it is called the appropriate method from the *interpolation* module; in linear and spline interpolation are used methods of *scipy.interpolate* that compute the whole set of points and return the interpolating function, which is then sampled:

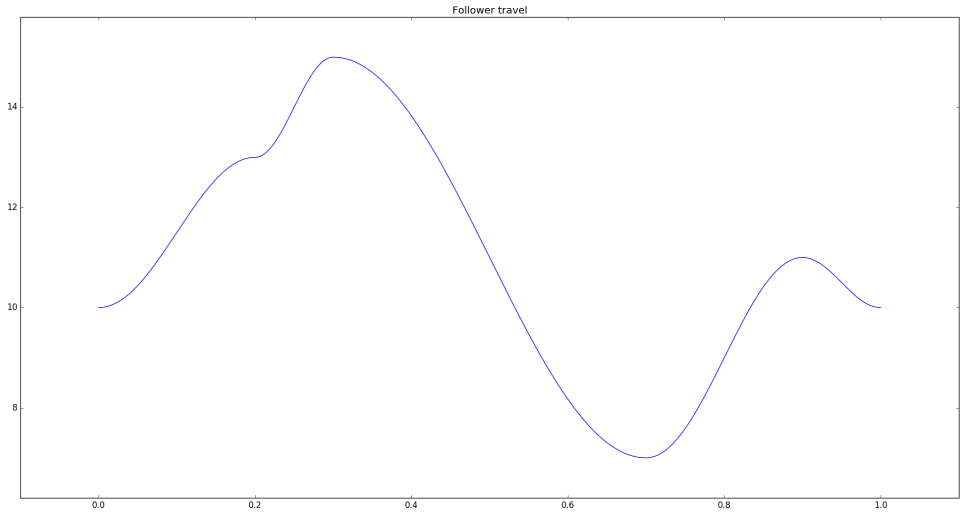
```
# Linear
f = interpolate.interp1d(xpoints, ypoints)
x = np.linspace(0, 1, steps)
y = f(x)

# Spline
tck = interpolate.splrep(xpoints, ypoints, k=order, per=True)
x = np.linspace(0, 1, steps)
y = interpolate.splev(x, tck)
```

In harmonic, cycloidal, parabolic and polynomial interpolation a linking function is found for every couple of consecutive points:

```
# Harmonic
x = np.linspace(x0, x1, steps)
y = y0 + (y1-y0)/2*(1-np.cos(np.divide(np.pi*(x+(-x0)), x1-x0)))

# Cycloidal
x = np.linspace(x0, x1, steps)
r = np.divide(x+(-x0), x1-x0)
y = y0 + (y1-y0)/np.pi*(np.pi*r-np.sin(2*np.pi*r)/2)

# Parabolic
x = np.linspace(x0, (x0+x1)/2, steps)
r = np.divide(x + (-x0), x1 - x0)
y = y0 + 2*(y1-y0)*r**2
x2 = np.linspace((x0+x1)/2, x1, steps)
r2 = np.divide(x2 + (-x0), x1 - x0)
y2 = y0 + (y1-y0)*(1-2*(1-r2)**2)

# Polynomial, solve Vandermonde matrix with added rows for conditions on derivatives
A = np.zeros((2+order*2, 2+order*2))
B = np.zeros(2+order*2)
B[0] = 0
B[1] = y1-y0
for col in range(0, 2+order*2):
```

```
    A[0, col] = 0 ** col
    A[1, col] = (x1-x0) ** col
for deriv in range(1, order + 1):
    B[deriv*2] = 0
    B[1+deriv*2] = 0
    for col in range(0, 2+order*2):
        # method der(x,n,k) returns the k-th derivative of x^n
        A[deriv*2, col] = der(0, col, deriv)
        A[1+deriv*2, col] = der((x1-x0), col, deriv)
c = np.linalg.solve(A, B)
x = np.linspace(0, x1-x0, steps)
y = np.polyval(c[::-1], x)
```

The result is kept in memory in the arrays *x* and *y* of a persistent instance of the class *Travel*.



4 Cam generation

```
usage: gen cam [-h] [--radius R] [--ccw] [--flat] [--offset OFF] [--fradius R]

optional arguments:
  -h, --help            show this help message and exit
  --radius R, -r R      base radius (default: 0)
  --ccw                 counterclockwise (default: False)
  --flat, -f            flat follower (default: False)
  --offset OFF, -d OFF  follower offset (default: 0)
  --fradius R, -q R     follower radius (set 0 for knife edge) (default: 0)
```

The cam is generated from the travel saved in memory. The class *Cam* distinguishes the follower kind and calls the appropriate methods. In case of a knife follower the cam profile is generated immediately, else an appropriate method is called.

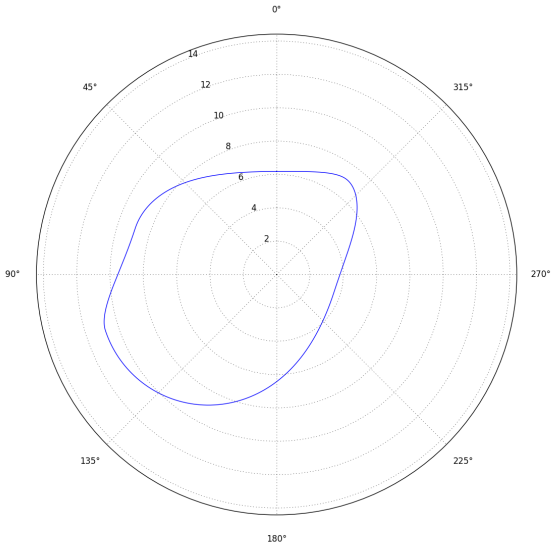
```
if follower.kind == 'knife':
```

```
    if follower.offset == 0:
        pcoords[1] = travel.y + radius
    else:
        pcoords[1] = np.sqrt(follower.offset**2+(radius+travel.y)**2)
elif follower.kind == 'roller':
    if follower.offset == 0:
        pcoords[1] = travel.y + radius - follower.radius
    else:
        rho_trace = np.sqrt(follower.offset ** 2 + (radius + travel.y) ** 2)
        pcoords = envelope.roller(pcoords[0], rho_trace, follower.radius)
elif follower.kind == 'flat':
    pcoords = envelope.flat(np.array([pcoords[0], travel.y + radius]))
if ccw:
    pcoords[1] = pcoords[1][::-1]
```

In the pseudo-envelope of flat follower the program, for each point of the pitch curve, calculates the distances of intersection of the lines passing through every other point and perpendicular to the line connecting the point to the origin ($\perp\ \theta$) and takes the smallest positive one as value of ρ in the point direction θ_0.

The roller follower envelope is done by offsetting of follower radius the polygonal pitch curve. For it is used Pyclipper, a wrapper of Clipper library, which only computes integers, so a scaling preserving a good precision is needed. Here is used a factor that sets the precision to the maximum error ϵ of the polygonial representation of a circle with radius equal to the maximum picth curve's ρ. $\epsilon = \frac{\rho_{max}}{1-cos(\pi/\delta\theta)}$
A confirmation of the goodness of this scaling is the adequate number of points returned by Clipper as result.

```
def flat(pcoords):
    result = np.empty_like(pcoords)
    for i, theta0 in enumerate(pcoords[0]):
        distances = pcoords[1] / np.cos(pcoords[0] - theta0)
        result[1][i] = np.min(distances[distances > 0])
    return result

def roller(thetas, rhos, radius):
    # Clipper works only with integers, scaling needed
    p = cartesian(thetas, rhos)
    scale = 1/(rhos.max()*(1-np.cos(np.pi/thetas.shape[0])))
    p *= scale
    coords = p.astype(int)
    pco = pyclipper.PyclipperOffset()
    pco.AddPath(coords, pyclipper.JT_ROUND, pyclipper.ET_CLOSEDPOLYGON)
    result = pco.Execute(-radius*scale)[0]
    p = polar(*zip(*result))
    p[1] /= scale
    return p
```



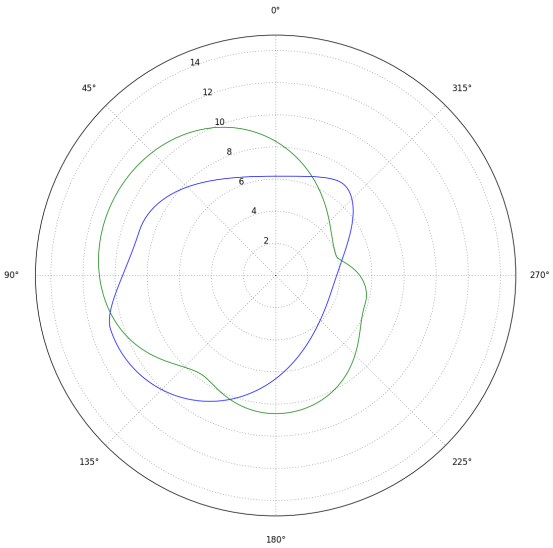
5 Conjugated cam generation

```
usage: gen conj [-h] [--breadth B]

optional arguments:
  -h, --help            show this help message and exit
  --breadth B, -b B     breadth, if 0 calculate optimal (default)
```

The program is able to generate the constant-breadth conjugated cam. The breadth can be set or else it is used the mean of cam diameters, an optimal value.

```
def gen_conjugated(self, breadth=None):
    # For flat follower and others with diametrically opposed followers
    if breadth is None:
        breadth = self.breadth
    if breadth == 0:
        # Set breadth = mean cam diameter
        print('Searching optimal breadth')
        diam = self.pcoords[1] + self.interp(self.pcoords[0]+np.pi)
        breadth = diam.sum() / self.pcoords.shape[1]
    elif breadth < np.max(self.pcoords[1]):
        # maybe could be, with opposed values sum > 0
        raise HandledValueError('Breadth must be >= max radius ({:.3g})'
                                .format(np.max(self.pcoords[1])))
    self.breadth = breadth
    print('Breadth:', breadth)
    self.conj_pcoords = np.empty_like(self.pcoords)
    for i in range(0, len(self.pcoords.T)):
        self.conj_pcoords[:, i] = (self.pcoords[0, i]+np.pi) % (2*np.pi),
                                   breadth - self.pcoords[1][i]
    # just fast, but probably a shift would be enough
```

```
self.conj_pcoords = self.conj_pcoords[:, self.conj_pcoords[0].argsort()]
```



6 STL exportation

```
usage:  export [-h] file width

positional arguments:
  file        output stl file
  width       cam width
```

The 3-d model is a surface built with triangles. The program builds the 2-d face with Delaunay triangulation, then adds s third coordinate valued 0 to build the lower face, and valued *width* for the upper face.

```
# 2D Delaunay triangulation for front face and building of lower and upper faces
tri0 = cam.points[Delaunay(cam.points).simplices]
tri1 = np.concatenate((tri0, np.zeros([tri0.shape[0], tri0.shape[1], 1])), 2)
tri2 = np.concatenate((tri0, np.ones([tri0.shape[0], tri0.shape[1], 1]) * width), 2)
```

Side surface is generated calculating each triangle's coordinates, which essentially links every couple of consecutive points of one face to a point on the other, as seen here:

```
# Build triangles for side face
vertices1 = np.concatenate((cam.points, np.zeros([cam.points.shape[0], 1])), 1)
vertices2 = np.concatenate((cam.points, np.ones([cam.points.shape[0], 1]) * width), 1)
tri3_1 = np.empty([cam.points.shape[0], 3, 3])
tri3_2 = np.empty_like(tri3_1)
for i in range(0, vertices1.shape[0]):
    tri3_1[i] = [vertices1[i-1], vertices1[i], vertices2[i]]
    tri3_2[i] = [vertices2[i-1], vertices2[i], vertices1[i-1]]
```

The program then concatenates the triangles and proceeds to save the model using methods from the *stl* library.

7 Simulation

```
usage:  sim [-h] [--omega W | --rpm RPM] [--steps S] [--precision P]
            [--gravity G]

optional arguments:
  -h, --help           show this help message and exit
  --omega W, -w W      angular velocity (default: 1)
  --rpm RPM, -r RPM    revolutions per minute (default: None)
  --steps S, -s S      steps (default: 500)
  --precision P, -p P  precision (default: 0.001)
  --gravity G, -g G    gravitational acceleration (default: 9.8)
```

The simulation is really basic and is just intended for visualization. The program draws the rotating cam and calculates the position of the follower. For flat follower the position is the maximum *y* of the rotated cam `max(cam_coords[i][1])`. For knife follower it calculates the intersection of the cam with a vertical line at the follower offset and takes the upper *y* bound `foll_body.intersection(cam_body).bounds[3]`. For roller follower the position is approximated, with given precision, with a bisection-like method. The first guess is found translating the follower with a constant step *s* until the *intersection state* changes, then halves *s* and translates in the opposite direction. This ensure that the state changes between *height-s* and *height+s*. The precision is then improved with a bisection loop, translating and halving *s* in each interarion. Finally the next height's first guess and *s* are calculated with the finite differential.

```
if foll_body.intersects(cam_body):
    translate(foll_body, yoff=s)
    height += s
    while foll_body.intersects(cam_body):
        translate(foll_body, yoff=s)
        height += s
    s /= 2
    translate(foll_body, yoff=-s)
    height -= s
else:
    translate(foll_body, yoff=-s)
    height -= s
    while not foll_body.intersects(cam_body):
        translate(foll_body, yoff=-s)
        height -= s
    s /= 2
    translate(foll_body, yoff=+s)
    height += s
s /= 2

while s >= precision:
    if not foll_body.intersects(cam_body):
        translate(foll_body, yoff=-s)
        height -= s
    else:
```

```
            translate(foll_body, yoff=s)
            height += s
        s /= 2

foll_heights[i] = height

if i > 0:
    diff = height - foll_heights[i-1]
    height += diff
    translate(foll_body, yoff=diff)
    s = abs(diff) if diff != 0 else precision
```

Once all follower heights are determined the program calculates velocity and acceleration, then renders and displays the animation.

https://github.com/fciotti/camdesign

8 Source code

main.py

```
import argparse
from travel import Travel
from cam import Cam
from follower import Follower
import matplotlib.pyplot as plt
import fileio
import simulation
import numpy as np
from utils import HandledValueError, plot_polar
import export


class ArgumentError(Exception):
    pass


# Overriding ArgumentParser error handling
class ArgumentParser(argparse.ArgumentParser):
    def error(self, message):
        raise ArgumentError(self, message)

    # Do not exit after printing help
    def exit(self, status=0, message=None):
        # if message:
        #     self._print_message(message)
        self.error(message)

interp = ['spline', 'linear', 'harmonic', 'cycloidal', 'parabolic', 'polynomial']
targets = ['travel', 'cam', 'conj']

parser = ArgumentParser(prog='', add_help=False)
subparsers = parser.add_subparsers(dest='command', prog='')
subparsers.required = True

parser_help = subparsers.add_parser('help', help='show this help message')
parser_exit = subparsers.add_parser('exit')

parent_travel = ArgumentParser(add_help=False)
parent_travel.add_argument('-k', dest='kind', choices=interp, help='kind of interpolation')
parent_travel.add_argument('--order', '-o', type=int, help='spline/polynomial order', metavar='O')
```

```
parent_travel.add_argument('-n', type=int, help='repetitions per cycle')
parent_travel.add_argument('--steps', '-s', type=int, help='interpolation steps', metavar='S')
parent_travel.add_argument('--x0', '-a', type=float, help='lower bound of function evaluation')
parent_travel.add_argument('--x1', '-b', type=float, help='upper bound of function evaluation')

parent_cam = ArgumentParser(add_help=False)
parent_cam.add_argument('--radius', '-r', type=float, help='base radius', metavar='R')
parent_cam.add_argument('--ccw', action='store_const', const=True, help='counterclockwise')
parent_cam.add_argument('--flat', '-f', action='store_const', const=True, help='flat follower')
parent_cam.add_argument('--offset', '-d', type=float, help='follower offset', metavar='OFF')
parent_cam.add_argument('--fradius', '-q', type=float, help='follower radius (set 0 for knife edge)', metavar='R')

parent_conj = ArgumentParser(add_help=False)
parent_conj.add_argument('--breadth', '-b', type=float, help='breadth, if 0 calculate optimal (default)', metavar='B')

parser_gen = subparsers.add_parser('gen', help='generate, unspecified variables set to default')
subparsers_gen = parser_gen.add_subparsers(dest='target')
subparsers_gen.required = True

parent_gen_travel_defaults = ArgumentParser(add_help=False)
parser_gen_travel = subparsers_gen.add_parser('travel', parents=[parent_travel],
                                              formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser_gen_travel_source = parser_gen_travel.add_mutually_exclusive_group(required=True)
parser_gen_travel_source.add_argument('--input', '-i', help='input file', metavar='IN')
parser_gen_travel_source.add_argument('--function', '-f', help='function of x', metavar='F')
parser_gen_travel.set_defaults(kind='linear', order=3, n=1, steps=10000, x0=0, x1=1)

parser_gen_cam = subparsers_gen.add_parser('cam', parents=[parent_cam],
                                           formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser_gen_cam.set_defaults(radius=0, ccw=False, flat=False, offset=0, fradius=0)

parser_gen_conj = subparsers_gen.add_parser('conj', parents=[parent_conj])
parser_gen_conj.set_defaults(breadth=0)

# Update has the same arguments as gen, but without overwriting with default values
parser_update = subparsers.add_parser('update', help='update, unspecified variables unmodified')
subparsers_update = parser_update.add_subparsers(dest='target')
subparsers_update.required = True
parser_update_travel = subparsers_update.add_parser('travel', parents=[parent_travel])
parser_update_cam = subparsers_update.add_parser('cam', parents=[parent_cam])
parser_update_cam.add_argument('--cw', dest='ccw', action='store_const', const=False, help='clockwise')
parser_update_cam.add_argument('--nonflat', dest='flat', action='store_const', const=False, help='non flat follower')
parser_update_conj = subparsers_update.add_parser('conj', parents=[parent_conj])

parser_load = subparsers.add_parser('load', help='load from file')
parser_load.add_argument('target', choices=['travel', 'cam'])
parser_load.add_argument('file', help='input file')

parser_save = subparsers.add_parser('save', help='save to file')
parser_save.add_argument('target', choices=targets)
parser_save.add_argument('file', help='output file')

parser_draw = subparsers.add_parser('draw', help='plot representation')
parser_draw.add_argument('targets', nargs='+', choices=targets)
parser_draw.add_argument('--unroll', '-u', action='store_true')
parser_draw.add_argument('--cartesian', '-c', action='store_true')

# parser_print = subparsers.add_parser('print', help='show current variables')

parser_export = subparsers.add_parser('export', help='export stl model')
parser_export.add_argument('file', help='output stl file')
parser_export.add_argument('width', type=float, help='cam width')
parser_export.add_argument('--conj', '-c', action='store_true')
```

```
parser_sim = subparsers.add_parser('sim', help='dynamic simulation',
                                   formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser_sim_velocity = parser_sim.add_mutually_exclusive_group(required=False)
parser_sim_velocity.add_argument('--omega', '-w', type=float, default=1, help='angular velocity', metavar='W')
parser_sim_velocity.add_argument('--rpm', '-r', type=float, help='revolutions per minute')
parser_sim.add_argument('--steps', '-s', type=int, default=500, help='steps', metavar='S')
parser_sim.add_argument('--precision', '-p', type=float, default=0.001, help='precision', metavar='P')
parser_sim.add_argument('--gravity', '-g', type=float, default=9.8, help='gravitational acceleration', metavar='G')

# TODO Simulation with conj, not simple
# TODO Export 2d polyline in DXF, ai, eps, pdf...

travel = Travel()
follower = Follower()
cam = Cam(travel, follower)

while True:
    try:
        command = input('camdesign: ').split()
        if len(command) and command[0] == 'update':
            # defaults = parser._defaults
            # parser._defaults = {}
            parser_gen_travel.set_defaults(kind=None, order=None, n=None, steps=None, x0=None, x1=None)
            parser_gen_cam.set_defaults(radius=None, ccw=None, flat=None, offset=None, fradius=None)
            parser_gen_conj.set_defaults(breadth=None)
            args = parser.parse_args(command)
            parser_gen_travel.set_defaults(kind='linear', order=3, n=1, steps=10000, x0=0, x1=1)
            parser_gen_cam.set_defaults(radius=0, ccw=False, flat=False, offset=0, fradius=0)
            parser_gen_conj.set_defaults(breadth=0)
            # parser._defaults = defaults
        else:
            args = parser.parse_args(command)

        # HELP
        if args.command == 'help':
            parser.print_help()

        # EXIT
        elif args.command == 'exit':
            exit()

        # GEN
        elif args.command == 'gen':
            if args.target == 'travel':
                travel.gen(args.input, args.function, args.x0, args.x1, args.kind, args.order, args.steps, args.n)
                if cam() and not cam.loaded:
                    print('Updating cam')
                    cam.gen()
            elif args.target == 'cam':
                if not travel():
                    print('Travel undefined')
                    continue
                follower.update(args.flat, args.offset, args.fradius)
                cam.gen(args.radius, args.ccw)
            elif args.target == 'conj':
                if not cam():
                    print('Cam undefined')
                    continue
                cam.gen_conjugated(args.breadth)

        # LOAD
        elif args.command == 'load':
            if args.target == 'travel':
                travel.load(args.file)
```

```
    elif args.target == 'cam':
        fileio.write(args.file, travel.x, travel.y)

# SAVE
elif args.command == 'save':
    if args.target == 'travel':
        if not travel():
            print('Travel undefined')
            continue
        fileio.write(args.file, travel.x, travel.y)
    elif args.target == 'cam':
        if not cam():
            print('Cam undefined')
            continue
        fileio.write(args.file, cam.pcoords[0], cam.pcoords[1])

# DRAW
elif args.command == 'draw':
    if 'travel' in args.targets:
        if not travel():
            print('Travel undefined')
            continue
        ax = plt.subplot(111)
        ax.margins(0.1)
        ax.plot(travel.x, travel.y)
        plt.title('Follower travel')
        plt.show()
    if 'cam' in args.targets or 'conj' in args.targets:
        if not cam():
            print('Cam undefined')
            continue
        plotted = []
        if 'cam' in args.targets:
            plotted.extend(cam.pcoords[:])
        if 'conj' in args.targets:
            if not cam.conj():
                # Maybe should call generation instead
                print('Conjugated cam undefined')
                continue
            plotted.extend(cam.conj_pcoords[:])
        if args.unroll:
            ax = plt.subplot()
            ax.margins(0.1)
            ax.plot(*plotted)
        else:
            if args.cartesian:
                ax = plt.subplot()
                ax.axis('equal')
                ax.margins(0.1)
                ax.plot(*plot_polar(*plotted))
            else:
                ax = plt.subplot(111, polar=True)
                ax.set_theta_zero_location('N')
                ax.plot(*plotted)
                ax.margins(0.5)
                ax.set_ylim(ymin=0)
        plt.show()

# UPDATE
elif args.command == 'update':
    if args.target == 'travel':
        if not travel.xpoints:
            print('Loaded cam, unable to update')
            continue
```

```
                print('Updating travel')
                travel.update(args.x0, args.x1, args.kind, args.order, args.steps, args.n)

                # If cam is generated (not loaded) repeat interpolation
                if cam() and not cam.loaded:
                    print('Updating cam')
                    cam.gen()
            elif args.target == 'cam':
                if not cam():
                    print('Cam not defined')
                    continue
                elif cam.loaded:
                    print('Loaded cam, unable to update')
                    continue
                follower.update(args.flat, args.offset, args.fradius)
                cam.gen(args.radius, args.ccw)
            elif args.target == 'conj':
                if not cam.conj():
                    print('Conjugated cam not defined')
                    continue
                cam.gen_conjugated(args.breadth)
        # SIMULATION
        elif args.command == 'sim':
            if not cam():
                print('Cam not defined')
                continue
            if args.rpm is not None:
                args.omega = args.rpm/60*2*np.pi
            simulation.draw(cam, follower, args.omega, args.steps, args.precision)
        # EXPORT
        elif args.command == 'export':
            export.stl(cam, args.file, args.width, args.conj)
    except (ArgumentError, argparse.ArgumentError, argparse.ArgumentTypeError) as ex:
        if isinstance(ex.args[0], ArgumentParser):
            if ex.args[1] is not None:
                print(ex.args[1])
                ex.args[0].print_usage()
        else:
            print(ex)
    except FileNotFoundError:
        print('File not found')
    except HandledValueError as ex:
        print(ex)
```

fileio.py

```
import argparse
from travel import Travel
from cam import Cam
from follower import Follower
import matplotlib.pyplot as plt
import fileio
import simulation
import numpy as np
from utils import HandledValueError, plot_polar
import export


class ArgumentError(Exception):
    pass


# Overriding ArgumentParser error handling
class ArgumentParser(argparse.ArgumentParser):
```

```
    def error(self, message):
        raise ArgumentError(self, message)

    # Do not exit after printing help
    def exit(self, status=0, message=None):
        # if message:
        #     self._print_message(message)
        self.error(message)

interp = ['spline', 'linear', 'harmonic', 'cycloidal', 'parabolic', 'polynomial']
targets = ['travel', 'cam', 'conj']

parser = ArgumentParser(prog='', add_help=False)
subparsers = parser.add_subparsers(dest='command', prog='')
subparsers.required = True

parser_help = subparsers.add_parser('help', help='show this help message')
parser_exit = subparsers.add_parser('exit')

parent_travel = ArgumentParser(add_help=False)
parent_travel.add_argument('-k', dest='kind', choices=interp, help='kind of interpolation')
parent_travel.add_argument('--order', '-o', type=int, help='spline/polynomial order', metavar='O')
parent_travel.add_argument('-n', type=int, help='repetitions per cycle')
parent_travel.add_argument('--steps', '-s', type=int, help='interpolation steps', metavar='S')
parent_travel.add_argument('--x0', '-a', type=float, help='lower bound of function evaluation')
parent_travel.add_argument('--x1', '-b', type=float, help='upper bound of function evaluation')

parent_cam = ArgumentParser(add_help=False)
parent_cam.add_argument('--radius', '-r', type=float, help='base radius', metavar='R')
parent_cam.add_argument('--ccw', action='store_const', const=True, help='counterclockwise')
parent_cam.add_argument('--flat', '-f', action='store_const', const=True, help='flat follower')
parent_cam.add_argument('--offset', '-d', type=float, help='follower offset', metavar='OFF')
parent_cam.add_argument('--fradius', '-q', type=float, help='follower radius (set 0 for knife edge)', metavar='R')

parent_conj = ArgumentParser(add_help=False)
parent_conj.add_argument('--breadth', '-b', type=float, help='breadth, if 0 calculate optimal (default)', metavar='B')

parser_gen = subparsers.add_parser('gen', help='generate, unspecified variables set to default')
subparsers_gen = parser_gen.add_subparsers(dest='target')
subparsers_gen.required = True

parent_gen_travel_defaults = ArgumentParser(add_help=False)
parser_gen_travel = subparsers_gen.add_parser('travel', parents=[parent_travel],
                                              formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser_gen_travel_source = parser_gen_travel.add_mutually_exclusive_group(required=True)
parser_gen_travel_source.add_argument('--input', '-i', help='input file', metavar='IN')
parser_gen_travel_source.add_argument('--function', '-f', help='function of x', metavar='F')
parser_gen_travel.set_defaults(kind='linear', order=3, n=1, steps=10000, x0=0, x1=1)

parser_gen_cam = subparsers_gen.add_parser('cam', parents=[parent_cam],
                                           formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser_gen_cam.set_defaults(radius=0, ccw=False, flat=False, offset=0, fradius=0)

parser_gen_conj = subparsers_gen.add_parser('conj', parents=[parent_conj])
parser_gen_conj.set_defaults(breadth=0)

# Update has the same arguments as gen, but without overwriting with default values
parser_update = subparsers.add_parser('update', help='update, unspecified variables unmodified')
subparsers_update = parser_update.add_subparsers(dest='target')
subparsers_update.required = True
parser_update_travel = subparsers_update.add_parser('travel', parents=[parent_travel])
parser_update_cam = subparsers_update.add_parser('cam', parents=[parent_cam])
parser_update_cam.add_argument('--cw', dest='ccw', action='store_const', const=False, help='clockwise')
parser_update_cam.add_argument('--nonflat', dest='flat', action='store_const', const=False, help='non flat follower')
```

```
parser_update_conj = subparsers_update.add_parser('conj', parents=[parent_conj])

parser_load = subparsers.add_parser('load', help='load from file')
parser_load.add_argument('target', choices=['travel', 'cam'])
parser_load.add_argument('file', help='input file')

parser_save = subparsers.add_parser('save', help='save to file')
parser_save.add_argument('target', choices=targets)
parser_save.add_argument('file', help='output file')

parser_draw = subparsers.add_parser('draw', help='plot representation')
parser_draw.add_argument('targets', nargs='+', choices=targets)
parser_draw.add_argument('--unroll', '-u', action='store_true')
parser_draw.add_argument('--cartesian', '-c', action='store_true')

# parser_print = subparsers.add_parser('print', help='show current variables')

parser_export = subparsers.add_parser('export', help='export stl model')
parser_export.add_argument('file', help='output stl file')
parser_export.add_argument('width', type=float, help='cam width')
parser_export.add_argument('--conj', '-c', action='store_true')

parser_sim = subparsers.add_parser('sim', help='dynamic simulation',
                                   formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser_sim_velocity = parser_sim.add_mutually_exclusive_group(required=False)
parser_sim_velocity.add_argument('--omega', '-w', type=float, default=1, help='angular velocity', metavar='W')
parser_sim_velocity.add_argument('--rpm', '-r', type=float, help='revolutions per minute')
parser_sim.add_argument('--steps', '-s', type=int, default=500, help='steps', metavar='S')
parser_sim.add_argument('--precision', '-p', type=float, default=0.001, help='precision', metavar='P')
parser_sim.add_argument('--gravity', '-g', type=float, default=9.8, help='gravitational acceleration', metavar='G')

# TODO Simulation with conj, not simple
# TODO Export 2d polyline in DXF, ai, eps, pdf...

travel = Travel()
follower = Follower()
cam = Cam(travel, follower)

while True:
    try:
        command = input('camdesign: ').split()
        if len(command) and command[0] == 'update':
            # defaults = parser._defaults
            # parser._defaults = {}
            parser_gen_travel.set_defaults(kind=None, order=None, n=None, steps=None, x0=None, x1=None)
            parser_gen_cam.set_defaults(radius=None, ccw=None, flat=None, offset=None, fradius=None)
            parser_gen_conj.set_defaults(breadth=None)
            args = parser.parse_args(command)
            parser_gen_travel.set_defaults(kind='linear', order=3, n=1, steps=10000, x0=0, x1=1)
            parser_gen_cam.set_defaults(radius=0, ccw=False, flat=False, offset=0, fradius=0)
            parser_gen_conj.set_defaults(breadth=0)
            # parser._defaults = defaults
        else:
            args = parser.parse_args(command)

        # HELP
        if args.command == 'help':
            parser.print_help()

        # EXIT
        elif args.command == 'exit':
            exit()

        # GEN
```

```
elif args.command == 'gen':
    if args.target == 'travel':
        travel.gen(args.input, args.function, args.x0, args.x1, args.kind, args.order, args.steps, args.n)
        if cam() and not cam.loaded:
            print('Updating cam')
            cam.gen()
    elif args.target == 'cam':
        if not travel():
            print('Travel undefined')
            continue
        follower.update(args.flat, args.offset, args.fradius)
        cam.gen(args.radius, args.ccw)
    elif args.target == 'conj':
        if not cam():
            print('Cam undefined')
            continue
        cam.gen_conjugated(args.breadth)

# LOAD
elif args.command == 'load':
    if args.target == 'travel':
        travel.load(args.file)
    elif args.target == 'cam':
        fileio.write(args.file, travel.x, travel.y)

# SAVE
elif args.command == 'save':
    if args.target == 'travel':
        if not travel():
            print('Travel undefined')
            continue
        fileio.write(args.file, travel.x, travel.y)
    elif args.target == 'cam':
        if not cam():
            print('Cam undefined')
            continue
        fileio.write(args.file, cam.pcoords[0], cam.pcoords[1])

# DRAW
elif args.command == 'draw':
    if 'travel' in args.targets:
        if not travel():
            print('Travel undefined')
            continue
        ax = plt.subplot(111)
        ax.margins(0.1)
        ax.plot(travel.x, travel.y)
        plt.title('Follower travel')
        plt.show()
    if 'cam' in args.targets or 'conj' in args.targets:
        if not cam():
            print('Cam undefined')
            continue
        plotted = []
        if 'cam' in args.targets:
            plotted.extend(cam.pcoords[:])
        if 'conj' in args.targets:
            if not cam.conj():
                # Maybe should call generation instead
                print('Conjugated cam undefined')
                continue
            plotted.extend(cam.conj_pcoords[:])
        if args.unroll:
            ax = plt.subplot()
```

```
                ax.margins(0.1)
                ax.plot(*plotted)
            else:
                if args.cartesian:
                    ax = plt.subplot()
                    ax.axis('equal')
                    ax.margins(0.1)
                    ax.plot(*plot_polar(*plotted))
                else:
                    ax = plt.subplot(111, polar=True)
                    ax.set_theta_zero_location('N')
                    ax.plot(*plotted)
                    ax.margins(0.5)
                    ax.set_ylim(ymin=0)
            plt.show()

    # UPDATE
    elif args.command == 'update':
        if args.target == 'travel':
            if not travel.xpoints:
                print('Loaded cam, unable to update')
                continue
            print('Updating travel')
            travel.update(args.x0, args.x1, args.kind, args.order, args.steps, args.n)

            # If cam is generated (not loaded) repeat interpolation
            if cam() and not cam.loaded:
                print('Updating cam')
                cam.gen()
        elif args.target == 'cam':
            if not cam():
                print('Cam not defined')
                continue
            elif cam.loaded:
                print('Loaded cam, unable to update')
                continue
            follower.update(args.flat, args.offset, args.fradius)
            cam.gen(args.radius, args.ccw)
        elif args.target == 'conj':
            if not cam.conj():
                print('Conjugated cam not defined')
                continue
            cam.gen_conjugated(args.breadth)
    # SIMULATION
    elif args.command == 'sim':
        if not cam():
            print('Cam not defined')
            continue
        if args.rpm is not None:
            args.omega = args.rpm/60*2*np.pi
        simulation.draw(cam, follower, args.omega, args.steps, args.precision)
    # EXPORT
    elif args.command == 'export':
        export.stl(cam, args.file, args.width, args.conj)
except (ArgumentError, argparse.ArgumentError, argparse.ArgumentTypeError) as ex:
    if isinstance(ex.args[0], ArgumentParser):
        if ex.args[1] is not None:
            print(ex.args[1])
            ex.args[0].print_usage()
    else:
        print(ex)
except FileNotFoundError:
    print('File not found')
except HandledValueError as ex:
```

```
        print(ex)
```

travel.py

```
import fileio
import interpolation
import numpy as np
from sympy import sympify
from sympy.utilities.lambdify import lambdify
from utils import HandledValueError
from sympy.abc import x


class Travel:
    x, y, xpoints, ypoints, levels, l, f, x0, x1, interp, order, steps, n =\
        None, None, None, None, None, None, None, None, None, None, None, None, None

    def __call__(self):
        return self.x is not None

    def __getattr__(self, name):
        if name == 'l':
            return not not self.levels  # True if levels are defied

    def load(self, filename):
        self.x, self.y = fileio.read(filename)
        self.xpoints, self.ypoints, self.levels, self.f = None, None, None, None

    def gen(self, filename, f, x0, x1, interp, order, steps, n):
        if filename is None:
            self.xpoints, self.ypoints, self.levels, self.f = None, None, None, None
            try:
                if f[0] == f[-1] == '"' or f[0] == f[-1] == "'":
                    f = f[1:-1]
                self.f = lambdify(x, sympify(f), modules="numpy")
            except:
                raise HandledValueError('Malformed expression')
        else:
            self.f = None
            self.xpoints, self.ypoints, self.levels = fileio.read(filename)
        self.update(x0, x1, interp, order, steps, n)

    def update(self, x0=None, x1=None, interp=None, order=None, steps=None, n=None):
        if x0 is not None:
            self.x0 = x0
        if x1 is not None:
            self.x1 = x1
        if interp is not None:
            self.interp = interp
        if order is not None:
            self.order = order
        if steps is not None:
            self.steps = steps
        if n is not None:
            self.n = n

        if self.xpoints is None:
            x = np.linspace(self.x0, self.x1, num=self.steps)
            try:
                y = self.f(x)
            except:
                raise HandledValueError('Malformed expression')
            self.x = np.linspace(0, 1, num=self.steps*self.n)
        else:
```

```
            # Warning if levels aren't respected
            if self.l and self.interp in ['spline']:
                print('Warning: levels not respected in', self.interp, 'interpolation')

            x, y = [], []
            if self.interp == 'spline':
                x, y = interpolation.spline(self.xpoints, self.ypoints, self.order, self.steps)
            elif self.interp == 'linear':
                x, y = interpolation.linear(self.xpoints, self.ypoints, self.steps)
                # x, y = self.xpoints, self.ypoints  # more sense, some problems with cam
            else:
                x, y = interpolation.points(self.interp, self.xpoints, self.ypoints, self.steps, self.order)

            # Fix x to [0;1]
            pts = [i for i, p in enumerate(x) if p >= 1]
            if len(pts) and self.interp not in ['linear', 'spline']:
                first = pts[0]  # find index of first point with x > 1
                x = [p-1 for p in x[first:]] + x[:first+1]
                y = y[first:] + y[:first+1]

            # Tile the profile
            self.x = np.empty([len(x)*self.n])
            for i in range(0, self.n):
                self.x.put(np.arange(i*len(x), (i+1)*len(x)), [(p+i)/self.n for p in x])  # increment x elements by i

        self.y = np.tile(y, self.n)
```

follower.py

```
class Follower:
    def __init__(self, flat=False, offset=0, radius=0):
        self.flat = flat
        self.offset = offset
        self.radius = radius

    def __getattr__(self, name):
        if name == 'kind':
            if self.flat:
                return 'flat'
            elif self.radius == 0:
                return 'knife'
            else:
                return 'roller'
        return None

    def update(self, flat, offset, radius):
        if flat is not None:
            self.flat = flat
        if offset is not None:
            self.offset = offset
        if radius is not None:
            self.radius = radius
```

interpolation.py

```
import numpy as np
from scipy import interpolate
from utils import HandledValueError


def spline(xpoints, ypoints, steps, order):
    if order > 5 or order >= len(xpoints):
        raise HandledValueError('Spline order must be smaller than number of points and not greater than 5')
```

```
    tck = interpolate.splrep(xpoints, ypoints, k=order, per=True)
    x = np.linspace(0, 1, steps)
    y = interpolate.splev(x, tck)
    return x, y


def linear(xpoints, ypoints, steps):
    f = interpolate.interp1d([xpoints[-2]-1, xpoints[-1]-1] + xpoints, [ypoints[-2], ypoints[-1]] + ypoints)
    x = np.linspace(0, 1, steps)
    y = f(x)
    return x, y


def points(kind, xpoints, ypoints, steps, order=None):
    x, y = [], []
    for i in range(0, len(xpoints) - 1):
        x0 = xpoints[i]
        y0 = ypoints[i]
        if kind == 'polynomial':
            xs, ys = func(kind, x0, xpoints[i + 1], y0, ypoints[i + 1], steps, order)
        else:
            xs, ys = func(kind, x0, xpoints[i + 1], y0, ypoints[i + 1], steps)
        x.extend(xs)
        y.extend(ys)
    return x, y


def func(kind, *args, **kwargs):
    if kind == 'harmonic':
        return harmonic(*args, **kwargs)
    elif kind == 'cycloidal':
        return cycloidal(*args, **kwargs)
    elif kind == 'parabolic':
        return parabolic(*args, **kwargs)
    elif kind == 'polynomial':
        return polynomial(*args, **kwargs)
    else:
        raise HandledValueError


def harmonic(x0, x1, y0, y1, steps):
    x = np.linspace(x0, x1, steps)
    y = y0 + (y1-y0)/2*(1-np.cos(np.divide(np.pi*(x+(-x0)), x1-x0)))
    return x, y


def cycloidal(x0, x1, y0, y1, steps):
    x = np.linspace(x0, x1, steps)
    r = np.divide(x+(-x0), x1-x0)
    y = y0 + (y1-y0)/np.pi*(np.pi*r-np.sin(2*np.pi*r)/2)
    return x, y


def parabolic(x0, x1, y0, y1, steps):
    x = np.linspace(x0, (x0+x1)/2, steps)
    r = np.divide(x + (-x0), x1 - x0)
    y = y0 + 2*(y1-y0)*r**2
    x2 = np.linspace((x0+x1)/2, x1, steps)
    r2 = np.divide(x2 + (-x0), x1 - x0)
    y2 = y0 + (y1-y0)*(1-2*(1-r2)**2)
    return np.concatenate((x, x2)), np.concatenate((y, y2))


def polynomial(x0, x1, y0, y1, steps, order):
```

```
    # Vandermonde matrix
    A = np.zeros((2+order*2, 2+order*2))

    B = np.zeros(2+order*2)

    B[0] = 0
    B[1] = y1-y0

    for col in range(0, 2+order*2):
        A[0, col] = 0 ** col
        A[1, col] = (x1-x0) ** col

    for deriv in range(1, order + 1):
        B[deriv*2] = 0
        B[1+deriv*2] = 0
        for col in range(0, 2+order*2):
            A[deriv*2, col] = der(0, col, deriv)
            A[1+deriv*2, col] = der((x1-x0), col, deriv)

    c = np.linalg.solve(A, B)

    x = np.linspace(0, x1-x0, steps)
    y = np.polyval(c[::-1], x)

    return x+x0, y+y0


# k-th derivative of x^n
def der(x, n, k):
    if k > n:
        return 0
    return np.math.factorial(n)/np.math.factorial(n-k)*x**(n-k)
```

cam.py

```
import numpy as np
import fileio
import envelope
from utils import cartesian, HandledValueError, Progress
from scipy.interpolate import interp1d


class Cam:
    pcoords, _points, f, rho_trace, spline, loaded, radius, ccw, conj_pcoords, breadth, width =\
        None, None, None, None, None, None, None, None, None, None, None

    def __getattr__(self, name):
        if name == 'theta':
            return self.pcoords[0]
        elif name == 'rho':
            return self.pcoords[1]
        elif name == 'ppoints':
            return self.pcoords.T
        elif name == 'points':
            if self._points is None:
                self._points = cartesian(*self.pcoords)
            return self._points
        elif name == 'coords':
            return self.points.T
        return None

    # Return cartesian coords after a theta0 counterclockwise rotation
    def rot_coords(self, theta0):
        return cartesian(self.theta + theta0, self.rho).T
```

```
# Return rho at given theta
def interp(self, theta):
    return self.f(theta % 2*np.pi)

def __call__(self):
    return self.pcoords is not None

def conj(self):
    return self.conj_pcoords is not None

def __init__(self, travel, follower):
    self.travel = travel
    self.follower = follower

def gen(self, radius=None, ccw=None):
    if radius is not None:
        self.radius = radius
    if ccw is not None:
        self.ccw = ccw
    self.loaded = False
    self._points = None
    self.conj_pcoords = None
    self.pcoords = np.empty([2, len(self.travel.x)])
    self.pcoords[0] = np.multiply(self.travel.x, 2 * np.pi)

    if self.follower.kind == 'knife':
        if self.follower.offset == 0:
            self.pcoords[1] = self.travel.y + self.radius
        else:
            self.pcoords[1] = np.sqrt(self.follower.offset**2+(self.radius+self.travel.y)**2)
    elif self.follower.kind == 'roller':
        if self.follower.offset == 0:
            self.pcoords[1] = self.travel.y + self.radius - self.follower.radius
        else:
            self.rho_trace = np.sqrt(self.follower.offset ** 2 + (self.radius + self.travel.y) ** 2)
            self.pcoords = envelope.roller(self.pcoords[0], self.rho_trace, self.follower.radius)
    elif self.follower.kind == 'flat':
        self.pcoords = envelope.flat(np.array([self.pcoords[0], self.travel.y + self.radius]))
    if self.ccw:
        self.pcoords[1] = self.pcoords[1][::-1]

    # Fixing x to [0;2pi], needed only for a few things
    self.pcoords[0] %= 2*np.pi
    # if self.pcoords[0][-1] == 0:
    #     self.pcoords[0][-1] = 2*np.pi
    self.pcoords = self.pcoords[:, self.pcoords[0].argsort()]

    # pts = [i for i, p in enumerate(self.pcoords[0]) if p >= 0]
    # if len(pts):
    #     first = pts[0]  # find index of first point with theta >= 0
    #     self.pcoords[0] = self.pcoords[0][first:] + self.pcoords[0][:first + 1]
    #     self.pcoords[1] = self.pcoords[1][first:] + self.pcoords[1][:first + 1]

    if(self.pcoords[0][0]) > 0:
        # Search last point with theta <= 2*pi and copy it to the start, shifted of -2*pi
        p = self.pcoords[:, self.pcoords[0] <= 2*np.pi][:, -1]
        self.pcoords = np.insert(self.pcoords, 0, [p[0] - 2*np.pi, p[1]], 1)
    if (self.pcoords[0][-1]) < 2*np.pi:
        # Search first point with theta >= 0 and copy it to the end shifted of 2*pi
        p = self.pcoords[:, self.pcoords[0] >= 0][:, 0]
        self.pcoords = np.insert(self.pcoords, -1, [p[0] + 2*np.pi, p[1]], 1)

    self.f = interp1d(*self.pcoords)
```

```
    def load(self, filename):
        self.pcoords = fileio.read(filename)
        self._points = None
        self.loaded = True
        self.conj_pcoords = None

    def gen_conjugated(self, breadth=None):
        # For flat follower and others with diametrically opposed followers
        if breadth is None:
            breadth = self.breadth
        if breadth == 0:
            # Set breadth = mean cam diameter
            print('Searching optimal breadth')
            diam = self.pcoords[1] + self.interp(self.pcoords[0]+np.pi)
            breadth = diam.sum() / self.pcoords.shape[1]
        elif breadth < np.max(self.pcoords[1]):
            # maybe could be, with opposed values sum > 0
            raise HandledValueError('Breadth must be >= max radius ({:.3g})'.format(np.max(self.pcoords[1])))
        self.breadth = breadth
        print('Breadth:', breadth)
        self.conj_pcoords = np.empty_like(self.pcoords)
        for i in range(0, len(self.pcoords.T)):
            self.conj_pcoords[:, i] = (self.pcoords[0, i]+np.pi) % (2*np.pi), breadth - self.pcoords[1][i]
        # just fast, but probably a shift would be enough
        self.conj_pcoords = self.conj_pcoords[:, self.conj_pcoords[0].argsort()]
```

envelope.py

```
import pyclipper
import numpy as np
from utils import cartesian, polar


# Super fast
def flat(pcoords):
    result = np.empty_like(pcoords)
    for i, theta0 in enumerate(pcoords[0]):
        distances = pcoords[1] / np.cos(pcoords[0] - theta0)
        result[1][i] = np.min(distances[distances > 0])
    return result


# Fast
def roller(thetas, rhos, radius):
    # Clipper works only with integers, scaling needed
    p = cartesian(thetas, rhos)
    scale = 1/(rhos.max()*(1-np.cos(np.pi/thetas.shape[0])))  # very good
    p *= scale
    coords = p.astype(int)
    pco = pyclipper.PyclipperOffset()
    pco.AddPath(coords, pyclipper.JT_ROUND, pyclipper.ET_CLOSEDPOLYGON)
    result = pco.Execute(-radius*scale)[0]
    p = polar(*zip(*result))
    p[1] /= scale
    return p
```

export.py

```
import numpy as np
from stl import mesh
from scipy.spatial import Delaunay
from matplotlib import pyplot as plt
```

```
from mpl_toolkits import mplot3d
from utils import cartesian


def stl(cam, filename, width, conj=False):
    if conj:
        points = cartesian(*cam.conj_pcoords)
    else:
        points = cam.points

    # 2D Delaunay triangulation for front face and building of lower and upper faces
    tri0 = points[Delaunay(points).simplices]
    tri1 = np.concatenate((tri0, np.zeros([tri0.shape[0], tri0.shape[1], 1])), 2)
    tri2 = np.concatenate((tri0, np.ones([tri0.shape[0], tri0.shape[1], 1]) * width), 2)

    # Build triangles for side face
    vertices1 = np.concatenate((points, np.zeros([points.shape[0], 1])), 1)
    vertices2 = np.concatenate((points, np.ones([points.shape[0], 1]) * width), 1)
    tri3_1 = np.empty([points.shape[0], 3, 3])
    tri3_2 = np.empty_like(tri3_1)
    for i in range(0, vertices1.shape[0]):
        tri3_1[i] = [vertices1[i-1], vertices1[i], vertices2[i]]
        tri3_2[i] = [vertices2[i-1], vertices2[i], vertices1[i-1]]

    # Concatenate and save
    tri = np.concatenate((tri1, tri2, tri3_1, tri3_2))
    data = np.zeros(tri.shape[0], dtype=mesh.Mesh.dtype)
    data['vectors'] = tri
    prism = mesh.Mesh(data)
    prism.save(filename)

    # Ugly plotting
    fig = plt.figure()
    ax = mplot3d.Axes3D(fig)
    ax.add_collection3d(mplot3d.art3d.Poly3DCollection(prism.vectors))

    # Auto scale
    scale = prism.points.flatten(-1)
    ax.auto_scale_xyz(scale, scale, scale)

    plt.show()
```

simulation.py

```
from matplotlib import pyplot as plt
from utils import *
from matplotlib.patches import Circle, Wedge
import numpy as np
from shapely.geometry import LinearRing, LineString
import matplotlib.animation as animation


def update(n, cam_plot, foll_plot, cam_coords, foll_heights, kind, steps, frames):
    i = int(n/frames*steps)
    cam_plot.set_data(cam_coords[i])
    if kind == 'flat':
        foll_plot.set_ydata((foll_heights[i], foll_heights[i]))
    elif kind == 'knife':
        foll_plot.set_center((foll_plot.center[0], foll_heights[i]))
    else:
        foll_plot.center = foll_plot.center[0], foll_heights[i]
    return cam_plot, foll_plot
```

```python
def translate(body, yoff):
    body.coords = [(point[0], point[1] + yoff) for point in body.coords]
    # for i in range(len(body.coords.xy[1])): slooow
    #     body.coords.xy[1][i] += yoff

    # arr = body.coords.array_interface()
    # np.ndarray(shape=arr['shape'], buffer=arr['data'])[:][1] += yoff  does not work (??)


# Not bad
def draw(cam, follower, omega, steps, precision):
    print('Initializing...')
    cam_points = cartesian(cam.theta, cam.rho)
    cam_body = LinearRing(cam_points).simplify(precision)
    height = np.max(cam.rho + follower.radius)
    rho_max = cam.rho.max()
    if follower.kind == 'knife':
        foll_body = LineString([(follower.offset, -rho_max), (follower.offset, rho_max)])
    elif follower.kind == 'roller':
        dt = 2*np.arccos(1-precision/follower.radius)
        foll_points = cartesian(np.arange(0, 2*np.pi, dt), follower.radius)
        foll_points[:, 0] += follower.offset
        foll_points[:, 1] += height
        foll_body = LinearRing(foll_points)
    cam_coords = np.empty([steps, 2, len(cam.theta)])
    foll_heights = np.empty([steps])

    if follower.kind == 'roller':
        s = follower.radius / 2
    else:
        s = cam.rho.min() / 2

    progress = Progress()
    for i, theta0 in enumerate(np.linspace(0, 2*np.pi, steps)):
        cam_coords[i] = cam.rot_coords(theta0)

        if follower.kind == 'roller':
            cam_body = LinearRing(cam_coords[i].T)  # maybe a rotation could be more efficient
            if foll_body.intersects(cam_body):
                translate(foll_body, yoff=s)
                height += s
                while foll_body.intersects(cam_body):
                    translate(foll_body, yoff=s)
                    height += s
                s /= 2
                translate(foll_body, yoff=-s)
                height -= s
            else:
                translate(foll_body, yoff=-s)
                height -= s
                while not foll_body.intersects(cam_body):
                    translate(foll_body, yoff=-s)
                    height -= s
                s /= 2
                translate(foll_body, yoff=+s)
                height += s
            s /= 2

            while s >= precision:
                if not foll_body.intersects(cam_body):
                    translate(foll_body, yoff=-s)
                    height -= s
                else:
                    translate(foll_body, yoff=s)
```

```
                height += s
            s /= 2

        foll_heights[i] = height

        if i > 0:
            diff = height - foll_heights[i-1]
            height += diff
            translate(foll_body, yoff=diff)
            s = abs(diff) if diff != 0 else precision
    elif follower.kind == 'flat':
        foll_heights[i] = max(cam_coords[i][1])
    else:
        cam_body = LinearRing(cam_coords[i].T)
        foll_heights[i] = foll_body.intersection(cam_body).bounds[3]
    # TODO calculate velocity and ...
    progress(theta0 / 2 / np.pi)
progress.close()

# Initialize graphics
fig, ax = plt.subplots(1, 2, sharey=True)
ax[0].axis('equal')
# max_x = max(max_coord, follower.offset % 1 + follower.radius)
# ax[0].set_xlim(-max_coord, max_x)
ax[0].set_ylim(-rho_max, np.max(foll_heights)+(follower.radius if follower.kind != 'flat' else 0))
ax[0].margins(1)
cam_plot, = ax[0].plot(*cam.coords)
if follower.kind == 'flat':
    # foll_plot = Rectangle((0, height), rho_max*2, 0)
    foll_plot = plt.Line2D((-rho_max, rho_max), (height, height), lw=rho_max/10)
    ax[0].add_line(foll_plot)
elif follower.kind == 'knife':
    foll_plot = Wedge((follower.offset, foll_heights[0]), rho_max, 89, 91)
    ax[0].add_patch(foll_plot)
else:
    foll_plot = Circle((follower.offset, foll_heights[0]), follower.radius, fill=False)
    ax[0].add_patch(foll_plot)
ax[0].plot()
ax[1].plot(np.linspace(0, 2*np.pi, steps), foll_heights)
t = 2*np.pi/omega
fps = 100
frames = round(fps*t)
fps_real = frames/t
interval = 1000/fps_real
anim = animation.FuncAnimation(fig, update, frames, interval=interval, repeat=True, blit=True,
                               fargs=(cam_plot, foll_plot, cam_coords, foll_heights, follower.kind, steps, frames))
plt.show()
```

utils.py

```
import numpy as np
import sys


class HandledValueError(ValueError):
    pass


def cartesian(theta, rho):
    x = rho * np.cos(theta)
    y = rho * np.sin(theta)
    return np.array([x, y]).T
```

```python
def polar(x, y):
    theta = np.arctan2(y, x)
    rho = np.sqrt(np.square(x) + np.square(y))
    return np.array([theta, rho])


def plot_polar(*args):
    result = []
    for i in range(0, len(args), 2):
        result.extend(cartesian(args[i], args[i+1]).T)
    return result


class Progress:
    def __init__(self):
        print('Progress: 0 %', end='')
        sys.stdout.flush()
        self.progress = 0

    def __call__(self, progress):
        prog = int(progress*100)
        if prog != self.progress:
            print('\rProgress:', prog, '%', end='')
            sys.stdout.flush()
        self.progress = prog

    def close(self):
        print('\rProgress: 100 %')
```