lab_geometry

March 30, 2024

1 lab geometry functions

the FullControl lab exists for things that aren't suitable for the main FullControl package yet, potentially due to complexity in terms of their concept, code, hardware requirements, computational requirements, etc.

FullControl features/functions/classes in the lab may be more experimental in nature and should be used with caution, with an understanding that they may change in future updates

at present, both the lab and the regular FullControl packages are under active development and the code and package structures may change considerably. some aspects currently in FullControl may move to lab and vice versa

lab currently has the following aspects: - geometry functions that supplement existing geometry functions in FullControl - four-axis and five-axis demos - transformation of a fullcontrol ***design*** into a 3D model (stl file) of the designed geometry with extudate heights and widths based on the designed `ExtrusionGeometry`

this notebook briefly demonstrates the geometry functions. four-axis, five-axis and 3d-model-output capabilities are demonstrated in their respective noetbooks: - 5-axis notebook - 4-axis notebook - stl-output notebook

FullControl lab import

[ ]:
```
import fullcontrol as fc
import lab.fullcontrol as fclab
from math import tau, radians
```

bezier curves

[ ]:
```
bez_points = []
bez_points.append(fc.Point(x=10, y=10, z=0))
bez_points.append(fc.Point(x=10, y=0, z=0))
bez_points.append(fc.Point(x=0, y=10, z=0))
bez_points.append(fc.Point(x=10, y=10, z=0))
bez_points.append(fc.Point(x=9, y=9, z=2))
steps = fclab.bezier(bez_points, num_points=100)
fc.transform(steps, 'plot', fc.PlotControls(style="line", zoom=0.8))
```

convex (streamline slicing) the CONVEX (CONtinuously Varied EXtrusion) approach allows continuously varying extrusion width. i.e. streamline-slicing see method images and case study in the associated journal paper (free download)

two outer edges are defined as paths and the CONVEX function fills the space between these edges with the desired number of paths

to travel between the end of one paths and start of the next, set `travel=True`

it optionally allows speed to be continuously matched to instantaneous extrusion width to maintain constant volumetric flowrate: - set `vary_speed` parameter to be True and supply values for `speed_ref` and `width_ref` parameters - these parameters are used to change speed proportional to how wide the instantaneous segment being printed is compared to width_ref

for open paths, it's useful to print lines using a zigzag strategy to avoid the nozzle moving back to the same side after printing each line - this is achieved by setting `zigzag=True`

set `overextrusion_percent` to achieve more extrusion while not changing the physical separation of line - this can help ensure good lateral bonding between neighbouring lines

it can be used to fill an arbirary shape by setting the 'inner edge' to be a list of coincident points near the centre of the shape. or for shapes with longish aspect ratios, 'inner edge' could be an abritrary list of points going along the appoximate medial axis and back.

these are example implementations, CONVEX can be used far more broadly

[ ]:
```
outline_edge_1 = fc.circleXY(fc.Point(x=0, y=0, z=0.2), 5, 0, 64)
outline_edge_2 = fc.circleXY(fc.Point(x=1, y=0, z=0.2), 3, 0, 64)
steps = fclab.convex_pathsXY(outline_edge_1, outline_edge_2, 10)
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence',␣
 ↪style='tube'))

# to vary speed to maintain constant flow rate:
# steps = fclab.convex_pathsXY(outline_edge_1, outline_edge_2, 10, vary_speed =␣
 ↪True, speed_ref = 1000, width_ref = 0.6)
```

[ ]:
```
outline_edge_1 = fclab.bezier([fc.Point(x=0, y=0, z=0.2), fc.Point(x=10, y=5,␣
 ↪z=0.2), fc.Point(x=20, y=0, z=0.2)], num_points = 100)
outline_edge_2 = fclab.bezier([fc.Point(x=0, y=10, z=0.2), fc.Point(x=10, y=5,␣
 ↪z=0.2), fc.Point(x=20, y=10, z=0.2)], num_points = 100)
steps = fclab.convex_pathsXY(outline_edge_1, outline_edge_2, 15, zigzag=True,␣
 ↪travel=False)
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence',␣
 ↪style='tube'))
```

instead of printing the above example geometry with lines running from end to end, you could design an imaginary 'outline' running along the medial axis of the specimen

then the part can be printed it in a similar manner to the ring example above, with lines printed from the outside inwards - there's just no hole in the middle

this is similar to concentric print paths in slicers, except the lines continuously fluctuate in width to achieve a shape-fitting path, which avoids islands needing to be printed with travel moves in between

[ ]:

```
points = 100
outline_edge_1 = fclab.bezier([fc.Point(x=0, y=0, z=0.2), fc.Point(x=10, y=5, z=0.2), fc.Point(x=20, y=0, z=0.2)], num_points = points)
outline_edge_2_reverse = fclab.bezier([fc.Point(x=20, y=10, z=0.2), fc.Point(x=10, y=5, z=0.2), fc.Point(x=0, y=10, z=0.2)], num_points = points)
inner_edge_1 = fc.segmented_line(fc.Point(x=4, y=5, z=0.2), fc.Point(x=16, y=5, z=0.2), points)
inner_edge_2_reverse = fc.segmented_line(fc.Point(x=16, y=5, z=0.2), fc.Point(x=4, y=5, z=0.2), points)
# create a closed path of the outline
outer_edge = outline_edge_1 + outline_edge_2_reverse + [outline_edge_1[0]]
# create a 'path' along the medial axis with the same number of points as 'outer_edge'
inner_edge = inner_edge_1 + inner_edge_2_reverse + [inner_edge_1[0]]
steps = fclab.convex_pathsXY(outer_edge, inner_edge, 7, travel=False)
steps.append(fc.PlotAnnotation(point = fc.Point(x=10, y=5, z=5), label="the default tube plotting style may not represent sudden changes in widths accurately"))
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence', style='tube'))
steps[-1] = (fc.PlotAnnotation(point = fc.Point(x=10, y=5, z=5), label="use fc.PlotControls(tube_type='cylinders') to get more accurate representations of printed widths"))
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence', style='tube', tube_type='cylinders'))
```

for arbitrary geometry with width-to-length aspect ratios approximately <2.5, it may be feasible to set the inner 'path' to be a list of identical points at a chosen centre point of the geometry

this example shows how CONVEX can fluctuate speed automatically to maintain constant volumetric flow rate

[ ]:

```
outer_edge = fclab.bezier([fc.Point(x=0, y=0, z=0),
                           fc.Point(x=20, y=0, z=0),
                           fc.Point(x=-10, y=6, z=0),
                           fc.Point(x=20, y=12, z=0),
                           fc.Point(x=0, y=12, z=0)], num_points=100)
outer_edge = fc.move_polar(outer_edge, fc.Point(x=0, y=6), 0, tau/2, copy=True, copy_quantity=2)
inner_edge = [fc.Point(x=0, y=6, z=0)]*len(outer_edge)
steps = fclab.convex_pathsXY(outer_edge, inner_edge, 12, travel=True, vary_speed=True, speed_ref=2000, width_ref=0.5)
widths_required = [step.width for step in steps if isinstance(step, fc.ExtrusionGeometry)]
speeds_required = [step.print_speed for step in steps if isinstance(step, fc.Printer)]
```

```
print(f'extrusion width varies from {min(widths_required):.2} to␣
↪{max(widths_required):.2} mm')
print(f'speed varies from {min(speeds_required)} to {max(speeds_required)} mm/
↪min, to maintain constant volumetric flow rate')
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence',␣
↪style='tube', tube_type='flow'))
```

offset a path required parameters:

- points: the supplied path - it must be a list of Point objects only
- offset: the distance to offset the path

optional parameters:

- flip: set True to flip the direction of the offset
- travel: set True to travel to the offset path without printing
- repeats: set the number of offsets paths - default = 1
- include_original: set True to return the original path as well as offset paths
- arc_outer_corners: set True to make outer corners have arcs (good for acute corners)
- arc_segments: numbers of segments par arc (if arc_outer_corners == True) - default = 8

[ ]:
```
points = [fc.Point(x=10, y=10, z=0.2), fc.Point(x=15, y=15, z=0.2), fc.
↪Point(x=20, y=10, z=0.2)]
offset = 0.4
steps = fclab.offset_path(points, offset, include_original=True, travel=True)
steps.insert(-2, fc.PlotAnnotation(label="the 'travel' parameter enables travel␣
↪movements to the start of offset paths"))
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence', zoom=0.
↪7, tube_type='cylinders'))
```

[ ]:
```
points = [fc.Point(x=10, y=10, z=0.2), fc.Point(x=15, y=15, z=0.2), fc.
↪Point(x=20, y=10, z=0.2), fc.Point(x=10, y=10, z=0.2)]
offset = 0.4
steps = fclab.offset_path(points, offset, include_original=True, travel=True)
steps.append(fc.PlotAnnotation(label="the offset path for a closed path is␣
↪automatically closed too"))
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence', zoom=0.
↪8, tube_type='cylinders'))
```

[ ]:
```
points = [fc.Point(x=10, y=10, z=0.2), fc.Point(x=15, y=15, z=0.2), fc.
↪Point(x=20, y=10, z=0.2), fc.Point(x=10, y=10, z=0.2)]
offset = 0.4
steps = fclab.offset_path(points, offset, travel=True, repeats=3,␣
↪include_original=True)
steps.insert(-2, fc.PlotAnnotation(label="path repeated multiple times using␣
↪the 'repeat' parameters"))
points2 = fc.move(points, fc.Vector(y=-7.5))
steps.extend(fc.travel_to(fc.Point(x=10, y=2.5)))
```

```
steps.extend(fclab.offset_path(points2, offset, travel=True, repeats=3, include_original=True, flip=True))
steps.insert(-2, fc.PlotAnnotation(label="path offset direction flipped using the 'flip' parameter"))
fc.transform(steps, 'plot',fc.PlotControls(color_type='print_sequence', zoom=0.8, tube_type='cylinders'))
```

[ ]:
```
points = [fc.Point(x=10, y=10, z=0.2), fc.Point(x=15, y=15, z=0.2), fc.Point(x=20, y=10, z=0.2), fc.Point(x=10, y=10, z=0.2)]
offset = 0.4
steps = fclab.offset_path(points, offset, repeats=10, travel = True, include_original=True, arc_outer_corners=True, arc_segments=16)
steps.append(fc.PlotAnnotation(label="add radii to corners with 'arc_outer_corners' and 'arc_segments' parameters"))
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence', tube_type='flow'))
```

reflect a list of points reflecting a list of points is complicated by the fact that the order in which controls are applied (e.g. to turn extrusion on or off) needs careful consideration - see more details about this in the regular geometry functions notebook

the following command can be used to reflect a list of points if it only contains points

[ ]:
```
steps = [fc.Point(x=0, y=0, z=0), fc.Point(x=1, y=1, z=0)]
steps += fclab.reflectXYpolar_list(steps, fc.Point(x=2, y=0, z=0), tau/4)
for step in steps: print(step)
```

find line intersection methods to find the intersection or to check for intersection between lines are demonstrated in the following code cell

[ ]:
```
line1 = [fc.Point(x=0, y=0, z=0), fc.Point(x=1, y=1, z=0)]
line2 = [fc.Point(x=1, y=0, z=0), fc.Point(x=0, y=1, z=0)]
intersection_point = fclab.line_intersection_by_points_XY(line1[0], line1[1], line2[0], line2[1])
print(f'\ntest 1... intersection at Point: {intersection_point}')

line_1_point = fc.Point(x=0, y=1, z=0)
line_1_angle = 0
line_2_point = fc.Point(x=1, y=0, z=0)
line_2_angle = tau/4
intersection_point = fclab.line_intersection_by_polar_XY(line_1_point, line_1_angle, line_2_point, line_2_angle)
print(f'\ntest 2... intersection at Point: {intersection_point}')

line1 = [fc.Point(x=0, y=0, z=0), fc.Point(x=1, y=1, z=0)]
line2 = [fc.Point(x=1, y=0, z=0), fc.Point(x=0, y=1, z=0)]
```

```
intersection_check = fclab.crossing_lines_check_XY(line1[0], line1[1],␣
↪line2[0], line2[1])
print(f'\ntest 3... intersection between lines (within their length):␣
↪{intersection_check}')
```

loop between lines 'loop__between__lines' allows smooth continuous printing between two lines - particularly useful for printing sacrificial material outside the region of interest for tissue engineering lattices, etc.

```
[ ]: line1 = [fc.Point(x=0, y=0, z=0.2), fc.Point(x=0, y=10, z=0.2)]
line2 = [fc.Point(x=10, y=10, z=0.2), fc.Point(x=20, y=0, z=0.2)]
loop_extension = 10 # dictates how far the loop extends past the lines
loop_linearity = 0 # 0 to 10 - disctates how linearly the loop initially␣
↪extends beyond the desired lines
loop = fclab.loop_between_lines(line1[0], line1[1], line2[0], line2[1],␣
↪loop_extension, travel=True, num_points=20, linearity=loop_linearity)
steps = line1 + loop + line2
fc.transform(steps, 'plot')
```

spherical coordinates spherical coordinates can be be used to define points with the fclab.spherical__to__point function

```
[ ]: point = fclab.spherical_to_point(origin = fc.Point(x=100, y=0, z=0), radius =␣
↪1, angle_xy=radians(90), angle_z = radians(0))
print(f'fclab.spherical_to_point() for angle_xy=90 degrees, angle_z = 0 degrees:
↪ \n    {repr(point)}')
point = fclab.spherical_to_point(origin = fc.Point(x=100, y=0, z=0), radius =␣
↪1, angle_xy=radians(90), angle_z = radians(45))
print(f'fclab.spherical_to_point() for angle_xy=90 degrees, angle_z = 45␣
↪degrees: \n    {repr(point)}')
point = fclab.spherical_to_point(origin = fc.Point(x=100, y=0, z=0), radius =␣
↪1, angle_xy=radians(90), angle_z = radians(90))
print(f'fclab.spherical_to_point() for angle_xy=90 degrees, angle_z = 90␣
↪degrees: \n    {repr(point)}')
```

fclab.spherical__to__vector is similar to fclab.spherical__to__point but does not need an origin to be defined since vectors can be considered to always be relative to xyz=0

the 'radius' parameter has also been replaced by the more logical term 'length'

```
[ ]: point = fclab.spherical_to_vector(length=10, angle_xy=radians(90),␣
↪angle_z=radians(45))
print(f'fclab.spherical_to_vector() for angle_xy=90 degrees, angle_z = 45␣
↪degrees: \n    {repr(point)}')
```

fullcontrol also has functions to determine spherical angles and radius from two points using fclab.point__to__spherical()

```
[ ]: point1 = fc.Point(x=10, y=0, z=0)
point2 = fc.Point(x=10, y=0, z=10)
spherical_data = fclab.point_to_spherical(origin_point=point1,␣
 ↪target_point=point2)
print(f'fclab.point_to_spherical() returns:\n    {repr(spherical_data)}')
angleZ_data = fclab.angleZ(start_point=point1, end_point=point2)
print(f'fclab.angleZ() returns:\n    {repr(angleZ_data)}')

recreated_point2 = fclab.spherical_to_point(point1,spherical_data.radius,␣
 ↪spherical_data.angle_xy, spherical_data.angle_z)
print(f'recreated point 2 using spherical data:\n    {repr(point2)}')
```

3D rotation rotate the toolpath or sections of the toolpath in 3D with fclab.rotate()

the function requires: - list of points - start point for the axis or rotation - end point for the axis of rotation (or 'x', 'y', or 'z') - angle of rotation - similar to fc.move(), if multiple copies are required: - copy = True - copy_quantity = number desired (including original)

```
[ ]: steps = fc.circleXY(fc.Point(x=10,y=0,z=0), 5,0)
steps = fclab.rotate(steps,fc.Point(x=30,y=0,z=0), 'y', tau/200, copy=True,␣
 ↪copy_quantity=75)
fc.transform(steps, 'plot', fc.PlotControls(zoom=0.7))
```

```
[ ]: start_rad, end_rad, EH = 3, 1, 0.4

bez_points = [fc.Point(x=0, y=0, z=0), fc.Point(x=0, y=0, z=10), fc.Point(x=10,␣
 ↪y=0, z=10),
              fc.Point(x=10, y=0, z=20), fc.Point(x=0, y=0, z=20), fc.
 ↪Point(x=-10, y=0, z=20)]
layers = int(fc.path_length(fclab.bezier(bez_points, 100))/EH)  # use 100␣
 ↪points to calculate bezier path length, then divide by extrusion height to␣
 ↪get the number of layers
centres = fclab.bezier(bez_points, layers)

radii = fc.linspace(start_rad, end_rad, layers)
segment_z_angles = [fclab.angleZ(point1, point2) if point2.x > point1.x else␣
 ↪-fclab.angleZ(point1, point2)
                    for point1, point2 in zip(centres[:-1], centres[1:])]
angles = segment_z_angles + [segment_z_angles[-1]]  # last point has now␣
 ↪segment after it, so use the angle of the previous segment

steps = []
for layer in range(layers):
    circle = fc.circleXY(centres[layer], radii[layer], 0, 64)
    circle = fclab.rotate(circle, centres[layer], 'y', angles[layer])
    steps.extend(circle + [fc.PlotAnnotation(point=centres[layer], label='')])
```

```
fc.transform(steps, 'plot', fc.PlotControls(style='line', zoom=0.4,␣
↪color_type='print_sequence'))
```