

design_tips

March 30, 2024

1 design tips

some anecdotal design considerations and examples / design-templates are provided here

<this document is a jupyter notebook - if they're new to you, check out how they work: [link](#), [link](#), [link](#)>

run all cells in this notebook in order (keep pressing shift+enter)

```
[ ]: import fullcontrol as fc
```

intellisense and auto-complete FullControl objects and functions take advantage of intellisense and auto-complete - in jupyter lab, use tab and shift+tab for auto-complete and intellisense - e.g. type 'fc.Poi' then tab then '(' then tab or shift+tab - in vscode, intellisense prompts pop-up automatically - check documentation for using intellisense in other software

use the 'travel_to' function for convenience the function fc.travel_to() is a convenient way to create a list of three steps: [Extruder(on=False), Point, Extruder(on=True)]

```
[ ]: steps = []
steps.append(fc.Point(x=0, y=0, z=0.2))
steps.append(fc.Point(x=5, y=5))
steps.extend([fc.Extruder(on=False), fc.Point(x=10, y=10), fc.
↳Extruder(on=True)])
steps.append(fc.Point(x=15, y=15))
steps.extend(fc.travel_to(fc.Point(x=20, y=20)))
steps.append(fc.Point(x=25, y=25))
fc.transform(steps, 'plot')
```

for-loops and fc.move() layers are sometimes simple repetitions of the layer beneath, in which case, simply copying the layer's steps with fc.move() is useful (case 1 in the code below)

a for-loop can be used instead (case 2)

this allows other factors to be freely changed (case 3)

```
[ ]: layers = 30

# case 1:
```

```

layer_steps = fc.rectangleXY(start_point=fc.Point(x=0,y=0,z=0.2), x_size=10,
    ↪y_size=10)
steps = fc.move(layer_steps, fc.Vector(z=0.2), copy=True, copy_quantity=layers)
steps.insert(-2, fc.PlotAnnotation(label='case 1'))

# travel to start of case 2
steps.extend(fc.travel_to(fc.Point(x=20,y=0,z=0.2)))

# case 2:
for i in range(layers):
    steps.extend(fc.rectangleXY(start_point=fc.Point(x=20,y=0,z=i*0.2),
    ↪x_size=10, y_size=10))
steps.insert(-2, fc.PlotAnnotation(label='case 2'))

steps.extend(fc.travel_to(fc.Point(x=40,y=0,z=0.2)))

# case 3: (x_size=10+i*0.2)
for i in range(layers):
    steps.extend(fc.rectangleXY(start_point=fc.Point(x=40,y=0,z=i*0.2),
    ↪x_size=10+i*0.2, y_size=10))
steps.insert(-2, fc.PlotAnnotation(label='case 3'))

fc.transform(steps, 'plot')

```

tau tau equals 2π

tau represents a full circle in radians, whereas pi represents half a circle

the natural unit of measure is generally a full circle, not half a circle

if you want an arc that is $\frac{3}{4}$ of a circle, arc length can be written as any of the following: 1.

arc_length = $0.75 * \text{tau}$ 1. arc_length = $1.5 * \text{pi}$ 1. arc_length = $0.75 * 2 * \text{pi}$

$0.75 * \text{tau}$ is more natural and clearer

consider the equivalent for units of years (similar to tau) or half-years (similar to pi). which of the following statements is clearer (all equivalent to the above three statements)?

1. I'm going on holiday in three quarters of a year
2. I'm going on holiday in one and a half half-years
3. I'm going on holiday in three quarters of two half-years

use tau!

```

[ ]: from math import tau
centre = fc.Point(x=0, y=0, z=0.2)
steps = fc.arcXY(centre, 10, 0, 0.75*tau)
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence'))

```

insert *state-changing* instructions retrospectively e.g. turn extrusion on/off after creating multiple copies of geometry

```
[ ]: steps = [fc.Point(x=i, y=i, z=0.2) for i in range(10)]
      steps.insert(4, fc.Extruder(on=False))
      steps.insert(8, fc.Extruder(on=True))
      fc.transform(steps, 'plot')
```

use `fc.PlotAnnotations()` and `fc.GcodeComments()` to debug and communicate designs annotating the 3D plot is incredibly useful for communicating design intention or changes to state that aren't easy to show in a 3D geometric plot (e.g. fan speed)

python's **f-strings** are a useful tool to generate annotation strings parametrically

```
[ ]: steps = []
      for i in range(13):
          steps.append(fc.Point(x=i+1, y=i+1, z=0))
          if i%2 == 0 and i<12:
              steps.append(fc.Fan(speed_percent=i*10))
              steps.append(fc.PlotAnnotation(label=f'fan speed {i*10}%'))
      fc.transform(steps, 'plot', fc.PlotControls(style='line',
          ↪color_type='print_sequence'))
```

inspecting gcode aside from opening the gcode file in a text editor or gcode-preview software, you can print a range of lines to screen, or use any of python's built-in functions to inspect the text

```
[ ]: output_type = 2 # change this to be 1, 2, or 3

      steps = [fc.Point(x=0,y=i,z=0) for i in range(11)]
      gcode = fc.transform(steps, 'gcode')
      gcode_list = (gcode.split('\n'))
      if output_type == 1:
          print(gcode)
      elif output_type == 2:
          print('\n'.join(gcode_list[5:8]))
      elif output_type == 3:
          for gcode_line in (gcode_list):
              if 'G1 F' in gcode_line or 'G0 F' in gcode_line:
                  print(gcode_line)
```

concise point and relative-point definition use P and R functions from the fullcontrol lab for concise definition of points in a design: - absolute points (P) -> `steps.append(P(x,y,z))` - relative points (R) -> `steps.append(R(x,y,z))`

`fc.lab.setup_p()` and `fc.lab.setup_r()` functions are used to create the P and R functions

the `setup_r()` function must be passed the variable you are using for your list of steps. relative points are always defined relative to the last point in that list

```
[ ]: import lab.fullcontrol as fclab

steps = []
P = fclab.setup_p()
R = fclab.setup_r(steps)

steps.append(P(40, 40, 0))
steps.append(R(0, 1, 0))
steps.append(R(1, 0, 0))

for step in steps: print(step)
```

new geometry functions create your own geometry functions

if you create useful geometry functions, add them to FullControl so everyone can benefit (see contribution guidelines on [github](#))

also see the section later in this notebook about using AI to generate geometric functions

```
[ ]: def saw_wave_x(start_point: fc.Point, length: float, amplitude: float, periods:
↳int) -> list:
    period_length = length/periods
    steps_wave = []
    for i in range(periods):
        steps_wave.append(fc.Point(x=start_point.x+period_length*i,
↳y=start_point.y, z=start_point.z))
        steps_wave.append(fc.Point(x=start_point.x+period_length*i,
↳y=start_point.y+amplitude, z=start_point.z))
        steps_wave.append(fc.Point(x=start_point.x+length, y=start_point.y,
↳z=start_point.z))
    return steps_wave

steps = []
steps.extend(saw_wave_x(fc.Point(x=20, y=20, z=0), 50, 10, 20))
steps.extend(saw_wave_x(steps[-1], 50, 20, 10))
steps.extend(saw_wave_x(steps[-1], 50, 10, 20))
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence'))
```

assemble *design-blocks* to create a *design*

```
[ ]: centre = fc.Point(x=0, y=0, z=0.2)
block1 = fc.spiralXY(centre, 0.5, 20, 0, 40, 2000)
block2 = fc.helixZ(centre, 20, 0, 0, 60, 0.3, 2200)
steps = block1 + block2
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence'))
```

use `fc.linspace()` to create list of evenly-spaced numbers

```
[ ]: print('e.g. "fc.linspace(0,1,5)": ' + str(fc.linspace(0,1,5)))
from math import tau
centre = fc.Point(x=0, y=0, z=0)
point_count = 100
radii = fc.linspace(10,20,point_count)
angles = fc.linspace(0,tau*2,point_count)
steps = [fc.polar_to_point(centre, radii[i], angles[i]) for i in
    range(point_count)]
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence'))
```

polar_coordinates points can be generated based on a polar-coordinates definition

state an origin, polar radius and polar angle

the first two points in the code below are identical but defined by different methods

```
[ ]: origin = fc.Point(x=0, y=0, z=0)
from math import tau

point_cart = fc.Point(x=10,y=0,z=0)
print(point_cart)
point_polar1 = fc.polar_to_point(origin, 10, 0)
print(point_polar1)
point_polar2 = fc.polar_to_point(origin, 10, tau/8)
print(point_polar2)
point_polar3 = fc.polar_to_point(origin, 10, tau/4)
print(point_polar3)
point_polar4 = fc.polar_to_point(origin, 10, -tau/4)
print(point_polar4)
```

vase mode polar coordinates allow vase mode to be achieved easily

```
[ ]: from math import cos, tau
layers = 50
segments_per_layer = 64
centre = fc.Point(x=50, y=50, z=0)
layer_height = 0.1
steps = []
for i in range(layers*segments_per_layer+1):
    # find useful measures of completion
    layer_fraction = (i%segments_per_layer)/segments_per_layer
    total_fraction = (int(i/segments_per_layer)+layer_fraction)/layers
    # calculate polar details
    angle = layer_fraction*tau
    radius = 5+1*cos(tau*(total_fraction))
    centre.z = layer_height*layers*total_fraction
    # add point
    steps.append(fc.polar_to_point(centre, radius, angle))
```

```
fc.transform(steps, 'plot', fc.PlotControls(zoom=0.8))
```

parametric maths equations with 't' (cartesian) use desmos to develop equations: [cartesian desmos link](#)

```
[ ]: from math import cos, tau
x_size = 20
y_offset, y_amplitude, waves = 5, 5, 3
t_steps = fc.linspace(0, 1, 101) # [0, 0.01, 0.02, ... , 0.99, 1]
steps = []
for t_now in t_steps:
    x_now = x_size*t_now
    y_now = y_offset+y_amplitude*cos(t_now*tau*waves)
    z_now = 0.2
    steps.append(fc.Point(x=x_now, y=y_now, z=z_now))
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence'))
```

parametric maths equations with 't' (polar) use desmos to develop equations: [polar desmos link](#)

```
[ ]: from math import cos, tau
centre = fc.Point(x=0, y=0, z=0)
inner_rad, rad_fluctuation, waves = 4, 1, 12
t_steps = fc.linspace(0, 1, 1001) # [0, 0.001, 0.002, ... , 0.999, 1]
steps = []
for t_now in t_steps:
    a_now = t_now*tau
    r_now = inner_rad+rad_fluctuation*cos(t_now*tau*waves)
    z_now = 0
    steps.append(fc.polar_to_point(centre, r_now, a_now))
steps = fc.move(steps, fc.Vector(z=0.1), copy=True, copy_quantity=10)
fc.transform(steps, 'plot')
```

‘post-process’ a *design or design-block* the following example creates a helical toolpath and then ‘post-processes’ it to change its geometry. the ‘post-process’ bit of code would work on different types of original geometry (e.g. a lattice-filled cylinder)

```
[ ]: # create a basic simple geometry (a cylinder) that will be modified
    ↪ retrospectively
centre = fc.Point(x=50, y=50, z=0)
steps = fc.helixZ(centre, start_radius=10, end_radius=10, start_angle=0,
    ↪ n_turns=50, pitch_z=0.167, segments=50*64)
steps.append(fc.PlotAnnotation(point = fc.Point(x=50, y=50, z=10),
    ↪ label='original geometry'))
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence', zoom=0.
    ↪ 7))
```

```

# 'post-process' the geometry to change it
z_max = 25
for step in steps:
    if isinstance(step, fc.Point):
        step.x -= 0.8*(step.x-centre.x)*(step.z/z_max)
        step.y -= 0.8*(step.y-centre.y)*(step.z/z_max)
        step.z -= (((step.y-centre.y)/2.5)**2)*(step.z/z_max)
steps[-1] = fc.PlotAnnotation(point = fc.Point(x=50, y=50, z=10),
    ↪label="'postprocessed' geometry")
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence', zoom=0.
    ↪7))

```

the following code cell shows a slightly more complex 'post-process', where a linear wave is created and then 'wrapped' around a cylinder to form an arc

this may be useful if it's easier to define some particular geometry in a linear format than in a curved format

```

[ ]: from math import tau
rad, rad_fluc, segs_per_period = 20, 5, 12
periods = 25
period_length = 3 # calculated to make sure the wave is the length of the
    ↪circle circumference
steps = fc.sinewaveXYpolar(fc.Point(x=rad, y=0, z=0.2), 0.75*tau, rad_fluc,
    ↪period_length, periods, segs_per_period)
steps.append(fc.PlotAnnotation(point=fc.Point(x=1.5*rad, y=-60, z=10),
    ↪label='original geometry'))
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence'))

def linear_to_arc(steps: list, centre: fc.Point, radius: float) -> list:
    '''this function assumes the linear geometry (list of points: 'steps') is
    ↪oriented in
        the y direction and positioned in the positive x-direction from the centre
    ↪point. it
        is 'wrapped' around an arc/circle dictated by radius. it is possible to
    ↪wrap multiple
        times. return list of translated points
    '''
    steps_wrapped = []
    for step in steps:
        rad_step = step.x - centre.x
        angle_step = (step.y - centre.y) / radius
        steps_wrapped.append(fc.polar_to_point(centre, rad_step, angle_step))
    return steps_wrapped

del steps[-1] # remove the PlotAnnotation
steps_wrapped = linear_to_arc(steps, fc.Point(x=0, y=0, z=0.2), 15)

```

```

steps_wrapped.append(fc.PlotAnnotation(point=fc.Point(x=0, y=0, z=10),
↳label="'post-processed' geometry"))

fc.transform(steps_wrapped, 'plot', fc.
↳PlotControls(color_type='print_sequence'))

```

generation speed don't worry too much about speed - future enhancements to the FullControl source code can easily:

- increase speed all-round
- allow quick previews
- include lightweight variants of objects with less functionality but greater speed
- explain how to generate designs quickly for specific **results** - e.g. no 'plot' option for `fc.transform()`, but greater gcode-generation speed
- explain how to create custom versions of FullControl for specific applications that only include strictly necessary functionality

immediate opportunities to increase the speed of plot previews:

- use **design-blocks** (described above) - create and preview them one at a time
- reducing segments for the 'plot' result but not the 'gcode' result
- increasing layer height to be unrealistically high for the 'plot' result but not the 'gcode' result
- create your design such that frequently changed **state**-changing objects that don't affect the plot are not created when creating a **design** (see the code-block below)
- for **designs** with lots of travel, create your design such that travel controls (`fc.Extruder(on=###)`) do not execute when creating a **design** for a 'plot' **result**
 - this means plotly plots the whole design as a single line series (fast), as opposed to lots of individual line series (slow)

```

[ ]: result_type = 'plot' # set as 'gcode' or 'plot'
steps = []
for i in range(6):
    if result_type != 'plot':
        # fan speed does not affect the 'plot' result so doesn't need to be
        ↳added to the design result_type == 'plot'
        steps.append(fc.Fan(speed_percent=100*i/5))
        steps.append(fc.Point(x=5*(i+1), y=5*((i+1)%2), z=0.2))
print(f"the design specifically for the '{result_type}' result contains
↳{len(steps)} steps:")
for step in steps: print(type(step).__name__)
fc.transform(steps, result_type, fc.PlotControls(color_type='print_sequence'))

```

use AI to create designs or geometry functions this is most suitable for users who have already got a good understanding of how to use FullControl since there is a reasonable risk that AI-generated code will not exactly match FullControl's requirements (e.g. it may generate a numpy array, or a dict instead of a list of FullControl Point objects)

example [chatGPT](#) request:

In python, a pydantic BaseModel class for a point has been imported in a module named fc

Create a list of points for a spiral with defined start radius, end radius, turns, number of points

the chatGPT output is given below. it is very sensitive to the question and can generate different responses when asked the same question twice. for the result generated below, chatGPT assumed the Point class was imported from the fc module rather than importing the whole fc module as done in these tutorial notebooks. therefore it doesn't use the fc.Point terminology, it just uses Point. a reasonable level of python knowledge is required to ensure the output can be tweaked as needed to work with FullControl as well as a good understanding of FullControl. E.g. this chatGPT code has no z values, but at least one point in your design must have a z value to allow a 3D plot

chatGPT output:

```
from math import cos, sin, pi
from fc import Point
from typing import List
```

```
def generate_spiral_points(start_radius: float, end_radius: float, turns: int, num_points: int):
    points = []
    radius_diff = end_radius - start_radius
    angle_step = 2 * pi * turns / num_points
    for i in range(num_points):
        radius = start_radius + radius_diff * i / (num_points - 1)
        angle = i * angle_step
        x = center.x + radius * cos(angle)
        y = center.y + radius * sin(angle)
        points.append(Point(x=x, y=y))
    return points
```

```
[ ]: from math import cos, sin, pi

def generate_spiral_points(start_radius: float, end_radius: float, turns: int,
    ↪ num_points: int, center: fc.Point) -> list:
    points = []
    radius_diff = end_radius - start_radius
    angle_step = 2 * pi * turns / num_points
    for i in range(num_points):
        radius = start_radius + radius_diff * i / (num_points - 1)
        angle = i * angle_step
        x = center.x + radius * cos(angle)
        y = center.y + radius * sin(angle)
        points.append(fc.Point(x=x, y=y, z=center.z)) # added 'fc.' and a z
    ↪ value is required for FullControl to create a plot in 3D space
    return points

steps = generate_spiral_points(10, 20, 4, 128, fc.Point(x=50, y=50, z=0))
```

```
fc.transform(steps, 'plot', fc.PlotControls(color_type='print_sequence'))
```