

一刀切り -One Complete Straight Cut- の理論と実践 3

3年F組6番 岩井宏朋

序論 Introduction

背景(Background) :

一昨年、昨年と取り組んできた一刀切り理論は、折り紙数学の一分野である。一刀切りを簡単に説明すると、1枚の折り紙を折り畳み、切り取りたい図形の輪郭線を一直線上に集めると、一回の切断により、その図形を切り出すことが出来るというものである。誰にも馴染みのある紙遊びで、折り紙を何度か折った後、一回だけハサミで切り、その後折り紙を開くと、花や動物、文字や図案が浮かび上がる。偶然にできた形状を楽しむのではなく、任意の図形の切り出しを意図して、この手順を逆から行うようにして考えるのが一刀切り理論の基礎的な考え方である。発展的には「一刀切りによってどれだけの形を作ることができるか」という問題が 1960 年に Martin Gardner によって提唱されていたが、この一刀切り問題は Erik D. Demain によって 1999 年に直線骨格法を用いて「ほとんど全ての場合」の証明がなされた。直線骨格法が満たせなかった問題点はディスクパッキング法という別のアルゴリズムによって本人により克服され、2000 年に「紙を折りたたみ一回切るだけで、どんな多角形でも無限に作れる」ことが数学的に証明された。

一刀切り理論を勉強することは空間認識能力を向上させ、同時に創造力や想像力を培うことができる。また、何より面白く、難解な図形の折り図は作るのに何度も試行錯誤を繰り返さなければならないので、幾何学の勉強方法としてとても価値が高いと思われる。

一昨年は折り紙数学の基礎定理と直線骨格法を学び K, E, I, O とそれぞれの文字を 1 文字ずつ手作業で折り図設計した。昨年は一刀切り理論への理解を更に深める事と KEIO の 4 文字を一つの多角形図形として折り図設計することを目標に取り組んだ。AutoCAD を用いる事で美しく折り目が連続するための図形の間隔の微調整なども容易になり、精密な設計をすることが出来た。本年は視点を変えて、任意の図形に対する一刀切りを可能にする折り図設計プログラムを作成することが目標である。

問題提起(Problem Presentations) :

一刀切り理論を考える上で、自動化（プログラム化）を行うためのアルゴリズムの設

計をすることは、一刀切り理論の数理の理解に非常に有益である。なぜなら、コンピューターが自分の代わりに様々な図形の折り図を作成してくれるため、考えついたアイデアの検証がスムーズに行えるようになるからだ。だが、公開されているものを検索した限りでは、一刀切りの自動設計図化アプリケーションは存在していなかった為、この制作に取り組みたい。

自動化が実現した場合、次のような社会貢献が期待できる。

1. 新たなアートスタイルの創造：

原来の折り紙アートは人間の独自の感性の元、作られていたが、自動化により、コンピュータを使用したアイディアを使う新たな表現方法が生まれる可能性がある。また昨年の経験から、いかなる図形も切り出し可能である事と美しく折って美しい形を得ることには乖離があることがわかった。いかなる図形でも折り目を増やせば最終的には輪郭辺を一直線上に揃えることは理論上可能だが、余りに折り目が多くなると実際の紙で折り重ねることは出来ない。理論上の紙には厚みが無く、現実の紙には厚みがあるからだ。

2. 教育：

折り紙は日本の誇る文化であり、数学の教材に取り入れることで幾何学の学習により親近感や面白みが加わるのではないかと思われる。背景で述べた通り、一刀切りは教育に適していると考えられ、一刀切り設計図の自動化により、様々な試行錯誤が容易になり、より直感的な理解をサポートするものになると期待できる。

3. 生産性の向上：

工業において、布などの畳みやすい素材を切る際に、本来は輪郭を切るが、自動化により切りたい形の折り図を作成し、一刀で済ませることが可能かもしれない。（折るプロセスと輪郭を切るプロセスのどちらの方がコストがかからないかは不明。また布の厚みの関係で、折った時に輪郭線にズレが生じる可能性もある。以上のことより、現段階では実現可能とは言えないが、コスト削減のアイデアの一つになる可能性がある）

実験目標(Experimental Goal)：

一刀切りの折り図を自動で作成してくれるアプリケーションを作り、正しいアルゴリズムが組み上げられているかを確認する。

方法 Materials and Methods

基盤技術と基礎知識の習得(Basic Technology and Knowledge)：

本研究では、システムの実装にあたって Python プログラミング、並びに画像情報の基礎を学んだ。Python をコンピュータ言語に選んだのは理由としては言語が英語の会話文に近く、図形の作図がしやすい言語だからである。またオブジェクト指向プログラミングという複雑なソフトウェアを構築するための方法論も一刀切りのアルゴリズムを書き起こす上で学んだ。

・Python プログラミング

1. CUI を用いた Python プログラムの実行：

CUI (Character User Interface) とはユーザーとコンピュータ間の情報のやり取りをする方法であり、本研究ではターミナルを用いた。

ファイルの操作

- ・現在位置の確認：pwd
- ・中に何があるかを確認：ls
- ・指定先に移動する：cd
- ・新しくディレクトリを作る：mkdir

上記のこととはターミナルで行う。

2. VSCode を用いた Python プログラムの編集：

VSCode (Visual Studio Code) はプログラミング言語を使用してソースコードを書くためのアプリであり、本研究では Python を使用した。

3. 基本構文：

if・else 文（もし～ならば、if 文の中身が、～以外なら else 文の中身が実行される）、for 文（簡単にいうと、for 文の中身をループさせることができる）、while 文（～である間は、while 文の中身が実行される）、変数[integer 型（文字列を整数に変換）、float 型（文字列を有効桁数の大きくない小数に変換）、boolean 型（真偽値(true or False)を保存する）、list 型（配列）、dictionary 型（辞書型:list がそれに対応する要素を持っている）]

4. モジュール：

モジュール（ライブラリともいう）とは関数の塊であり、コーディングをするにあたり、関数を 1 から全部タイピングするのは非合理的なので、ライブラリという便利なプログラムをまとめて一つのファイルにしたものを作成してインストールして使う。元々 Python 内に付属品としてついているもの（内部ライブラリ）だけでは足りないので、外部ライブラリという他の人が作ったライブラリも

使用させてもらう必要があった。一方で、自分で新たに作る場合は自作モジュールとして、コンピュータに保存して、別のファイルでも使用可能になる。

・オブジェクト指向プログラミング

1. オブジェクト指向プログラムにおける重要な概念、キーワード：

Object:データとそれに関連する操作をカプセル化したもの

Class:変数と関数をまとめたもので、共通の特性を持つオブジェクトの設計図やテンプレートとして機能する

Instance:クラスをもとに作られたオブジェクトのこと

`self`: class 内でオブジェクト自身のこと：my と同義

`__init__`: オブジェクトの初期化を決定づけている関数

以下の指標に沿って、一刀切り理論のプログラム化に必要な材料を書いていく。

・工場(モジュール)とオペレータ(実行用ファイル：run.py)

(プログラム上で欲しい関数を作る工場(自作モジュール的なもの)と作った関数を使って、アルゴリズムを書き上げるオペレータ(実行用ファイル))

2. 画像の形式：

コンピュータ上の画像には2種類のタイプがある。ラスター画像とは画質が良く写真などに利用されるが、線を拡大していくとギザギザになる。一方でベクター画像は画質があまり良くないが、拡大しても線のままである。ベクター画像は幾何学的図形などを描画するのによく使われる。よって本研究ではベクター形式の SVG 画像というものを使う。

提案手法(Proposed Methods)：

システムを構築するにあたって、全体の流れを先に説明する。

システム概要：

任意の図形を SVG 画像で入力（インプット）すると、その画像の要素を全て解析し、折り線を元の画像に描いて、編集後の画像をアウトプットとして返してくれるアプリを作成する。（コードの詳しい解説は付録に記載）

また実装したい一刀切りのアルゴリズムは、これまで取り組んできた一刀切りの折り図設計の経験から直線骨格法の理論に基づき手作業で進めた手順を再現することをまず目指すこととする。これを以下に記す。

1- 任意の多角形を設定する。

- 2- すべての角から二等分線を引く。
- 3- [仮定] 隣り合う二つの角を $\angle A \cdot \angle B$ と置く。そして、 $\angle B$ は線分ABと線分BCによってできる図形だとする。
- 3α - $\angle A \cdot \angle B$ の両方が劣角である時、 $\angle A \cdot \angle B$ の二等分線が設定した多角形内で交わる。交点から線分ABに向かって垂線を降ろす。
- 3β - $\angle A \cdot \angle B$ の片方が優角である時（ここでは $\angle B$ が優角だったとする）、 $\angle A \cdot \angle B$ の二等分線が設定した多角形内で交わる。交点から線分BCに向かって垂線を降ろす。
- 3γ - $\angle A \cdot \angle B$ の両方が優角である時、 $\angle A \cdot \angle B$ の二等分線が設定した多角形外で交わる。交点から線分ABに向かって垂線を降ろす。
- ($3\alpha \cdot 3\beta \cdot 3\gamma$ で引いた二等分線を山折りとする場合、ここで降ろした垂線は必ず谷折りである)
- (三角形の時は内心として、二等分線が一点で交わるので、特例として、垂線は一度しか降ろさない)
- 4- 交点同士を結ぶ（多角形の頂点の数が5以上の時の折り方については考察にて説明する）

一刀切り問題の証明をした Erik D. Demain は、証明にあたり直線骨格法の手順を一般化するアルゴリズムを考案した。簡易的に説明すると、任意の多角形について各辺から並行に等距離に図形を内側に縮小させていき、各頂点の軌跡をたどることで直線骨格を得る。その後、平坦折りを可能にするために、全ての骨格頂点から元の多角形の辺に向けて垂線を全て下ろし、この中から折り線として有効なものを選択していくという手順である。しかし、このアルゴリズムはほぼ全てのあらゆる多角形に対して一刀切りの可能性を証明するため、実際の条件設定は非常に複雑であることが予想される。尚、直線骨格法で証明できない例外的な事象を補うため、ディスクパッキング法によるアルゴリズムがあるが、これは更に複雑で折り線が多発されてしまう。本研究では、単純多角形についての手作業のアルゴリズムをまず実装し、その実験過程から一刀切り理論の数理的理解を深め、より複雑な図形への対応を目指していくこととする。参考に直線骨格法を一般化するアルゴリズム（簡易版）を以下に記す。

- 1- 与えられた図形の内側に各辺からの距離が $d > 0$ となる図形を描き、 d を段々大きくしていく。この時の頂点の軌跡が折り目となる。初めて1を実行する時のみ折り目を図形の外側にも延長する。
- 2- 1の図形が点または線分の集まりになった時、線分を折り目に加え、4に進む。
- 3- 1の図形が2つ内部を持つ多角形、または線分に分かれた時、線分は折り目とし、内部を持つ多角形についてはそれぞれ1の操作に戻る。

- 4- 折り目からなる図形の各頂点から与えられた多角形の辺に垂線を可能な限り下ろし、折り目とする。この折り目も図形の外側に延長する。ただし垂線と1から3の折り目が交わった時は、1から3の折り目にに対して線対称となるように曲げて垂線を延長する。この反射プロセスは垂線が骨格頂点にぶつかるか無限遠に行くまで繰り返される。
- 5- 1から3の折り目は必ず折り、4の折り目は折っても折らなくても良いという条件の下で正しい解を探す。

SVG 画像の編集（工場①）：

1. 書き込み

input : 任意の頂点群

output : 折り紙の枠線と頂点で指定された多角形を含む SVG 画像

ここでは、頂点の座標から多角形の辺を描画し、多角形というオブジェクトとして認識できるようにした。

2. 読み取り

input : SVG 画像

output : 画像に含まれる線を折り方[色（黒：折り紙の枠線、緑：図形の輪郭線、赤：山折り線、青：谷折り線）]で分別して獲得する
与えられた SVG 画像内の線を解析してそれぞれの線を分けて保管している（この情報を `svg_info` とした）。

3. 図形の追加

input : SVG 画像

process : 読み取り ->`svg_info` に情報（折り線）を追加する -> 書き込み

output : SVG 画像

書き込みは `svg_info` を元にして、SVG 画像を作成する。この process では、元の図形に折り線を引いていくが、欲しい折り線を引くモジュールは存在しないので、工場②で作り上げる。

多角形オブジェクトの作成（工場②）：

1. Point オブジェクト（class）

input : 自身の点座標とそれに隣接する点の座標

output : 自身の点と別の点の間の距離

[厳密には input、output では無く (class は関数ではない)、頂点が与えられた時に欲しい情報 (output) とそれを求めるのに必要な条件 (input) である]

ここでは、多角形 (頂点群) から全ての辺の長さを計算している (この時、長さはベクトルで表記される)。

2. Segment オブジェクト (class)

input : 線分の始点と終点の座標

output :

(1) 直線の式の一般形を求める関数

input : 自身の線分 (始点と終点の座標をもつ)

output : $ax+by+c=0$ の a, b, c

(2) 交点を求める関数

input : 自身の線分と別の線分

output : 交点の座標と自身の線分の始点から交点までの距離

(3) 垂線を降ろす関数

input : 自身の線分と任意の点

output : 任意の点から自身の線分へと下ろした垂線の式

3. Polygon オブジェクト (class)

input : 多角形 (各頂点の座標)

output :

(1) 線分 (辺) を作る関数

input : 全頂点の座標

output : 全ての辺の Segment オブジェクト

(2) 角の二等分線を引く関数

input : 多角形とリストになっている頂点群の位置 (何番目のデータなのかを表す = どの頂点を指定するかを決める)

process : ベクトルの概念を使い二等分線を引く

output : 二等分線の Segment オブジェクト

(3) 複数の角の二等分線を引く関数

input : 多角形

process : Segment オブジェクトで作った関数を利用し、角の二等分線から直線骨格を作っている。

output : 全ての角の二等分線と二等分線同士（隣り合う）の交点、その交点から降ろした垂線（それぞれの要素はこれまでに作った各関数の output に対応している 例：垂線なら垂線の式が output として返されている）

GUI の実装 :

GUI (Graphical User Interface) は CUI と似ており、CUI は文字、GUI は画像を用いる。GUI は絵を表示でき、マウスなどで画面を操作できるため、作図には非常に適している。任意の図形を作図する上で、各点の座標を入力していくは非効率であり大変なので、GUI を使うことによって、画面上のキャンバス（ウィンドウ）上に自由に作図できるようになる。

本研究では PySimpleGUI という Python で GUI を簡単に作るための外部ライブラリ（≒モジュール）を使用した。

1. 図形の描画

input : マウスなどのモーション（pad の移動、クリック等）

process : マウスの現在地と最後にクリックした場所を繋ぐ線を更新（線を‘引いては消して’を繰り返せば、最新の線しか残らない）し続ける

output : 閉じている多角形の画像

原理としてはマウスの移動から線を更新し続け、クリックした時に線を確定させれば線を消すことなく次の線を最後にクリックした場所から描くことが可能になる。（図 1 参照）

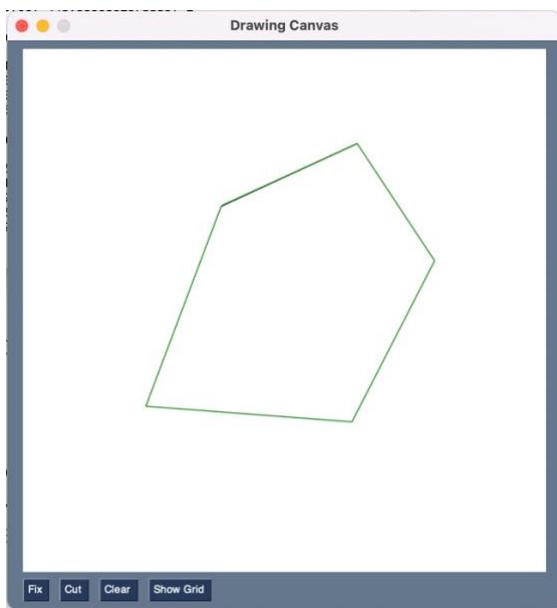


図1 五角形の描画

2. 図形の保存、折り線の追加

input：“Fix”->”Cut”または“Cut”というボタンを押す

process：工場①、②で作った関数を使用し折り線を追加

output：折り線が追加された新しいウィンドウ（画面）を表示

“Fix”や“Cut”的他にも“Clear”ボタン等も PySimpleGUI では作れる（機能は付録で説明）

3. グリッド機能

“Show Grid”と“Hide grid”を使い、ウィンドウにグリッドを表示したり、消したりすることを可能にした。また、点は最も近い格子状の点に引き寄せられる仕組みとし、正多角形のような精度が必要な図形の作図をしやすいようにした。（図2参照）

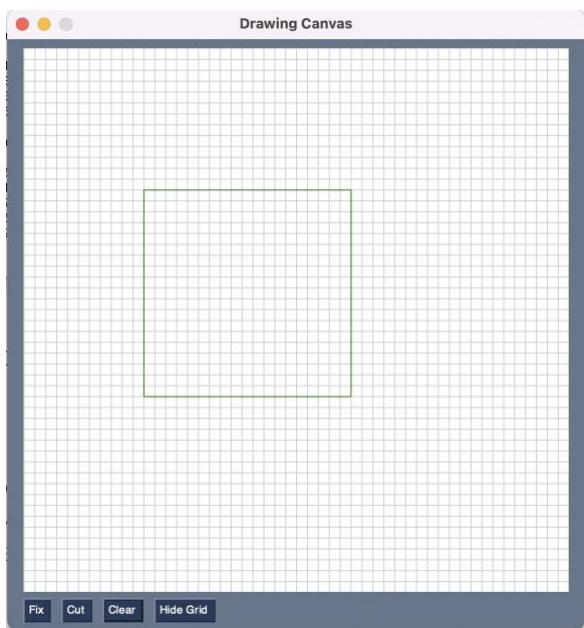


図2 グリッドを使った正方形の作図

実験環境 (Experimental Environment) :

デバイス情報 :

Apple M1 CPU8コア、RAM 8GB を搭載した MacBook Air を使用した。

OS 情報 :

Mac OS Monterey (ver.12.4)

コンピュータ言語 :

実験で使用したシステムは全て python (ver.3.11.4)を用いて作成した

外部モジュール :

lxml (4.9.3)
numpy (1.25.2)
PySimpleGUI (4.60.5)
svgwrite (1.4.3)
を使用した

実験 (Experiment) :

現段階で出来上がったアプリケーション（プログラム）を用いて、いろいろな多角形に折り線を引いてみる。

具体例：

三角形、凸四角形、凹四角形、五角形、etc.

結論 Conclusion

実験結果：

三角形は図3、凸四角形は図4、凹四角形は図5、対称性のある凹四角形は6、正方形は図7、菱形は図8、五角形は図9のような結果が得られた。

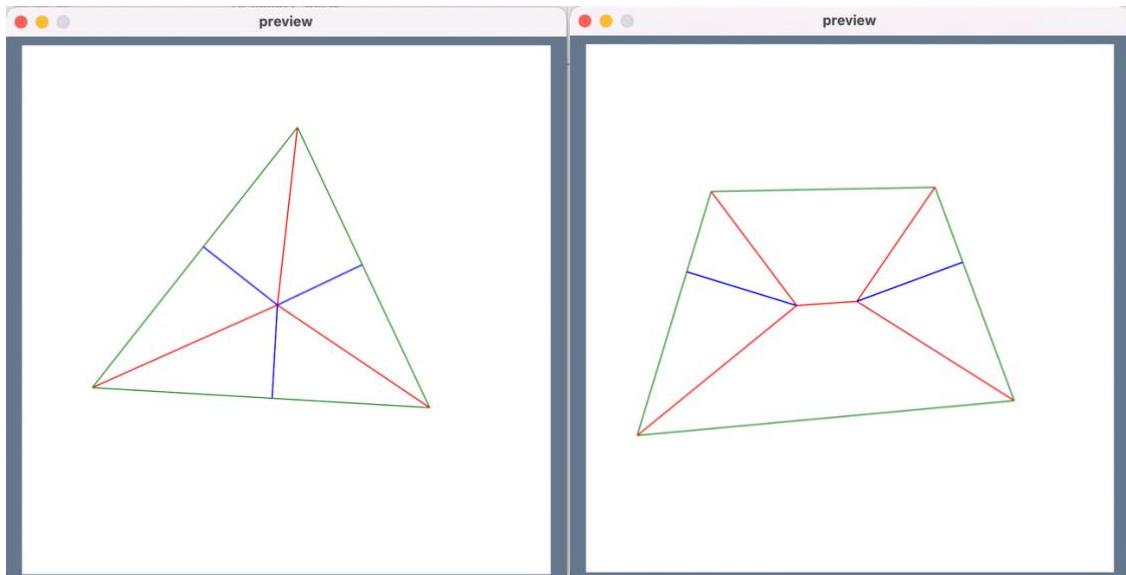


図3 三角形の実験結果

図4 凸四角形の実験結果

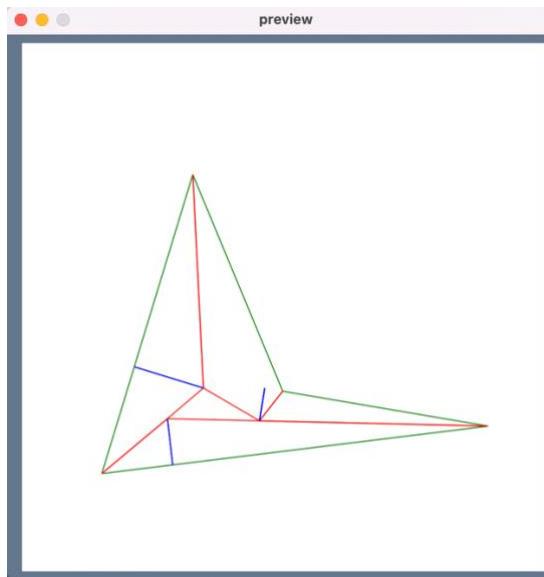


図 5 凸四角形の実験結果

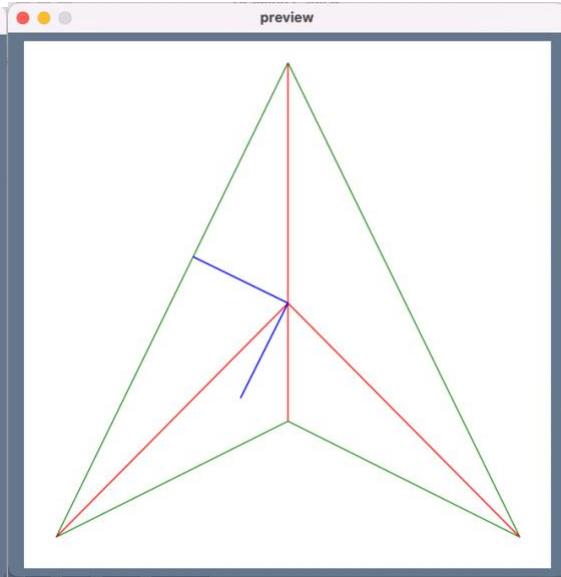


図 6 対称性のある凸四角形の実験結果

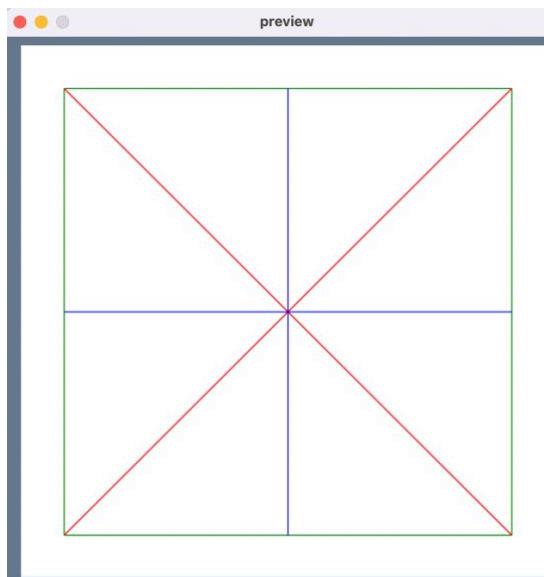


図 7 正方形の実験結果

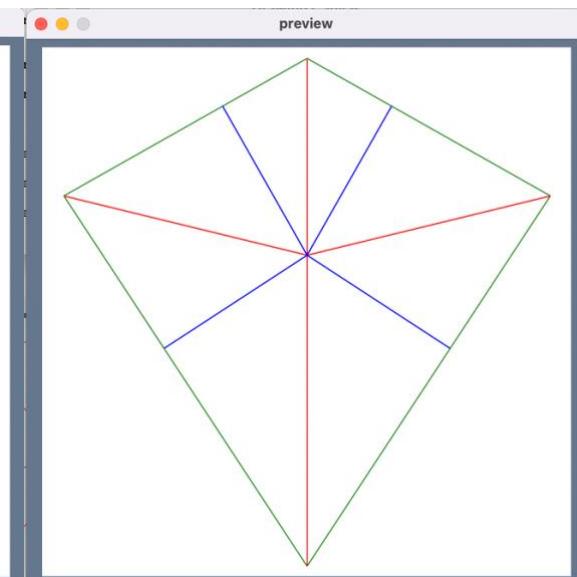


図 8 菱形の実験結果

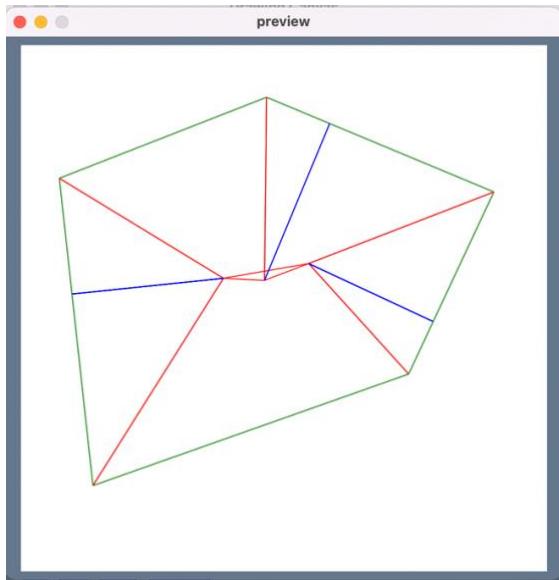


図9 五角形の実験結果

考察 :

三角形の折り図は本来垂線が1本のみであるはずだが、結果では3本も引かれてしまっている。垂線を引くコードに問題があると考えられる。垂線を引くコードは、ある頂点Aの二等分線と隣り合う二つの角の二等分線のうち先に交わる方(選ばれた頂点を頂点Bと仮定)を選んで、その交点から辺ABに垂線を降ろすようなプログラムにしている。三角形では全ての二等分線が一点で交わるため全辺に垂線が降ろされてしまうのだと考えられる。解決策としては、場合分けを行う。図形に折り線を書き加えるときに、if文で頂点の数が3つの時とそれ以外の時で場合分けを行う。3つの時は垂線を一邊だけに降ろせというコードに変更することで、いかなる三角形でも折り図の作成は可能になると考える(図10は完成した折り図)。

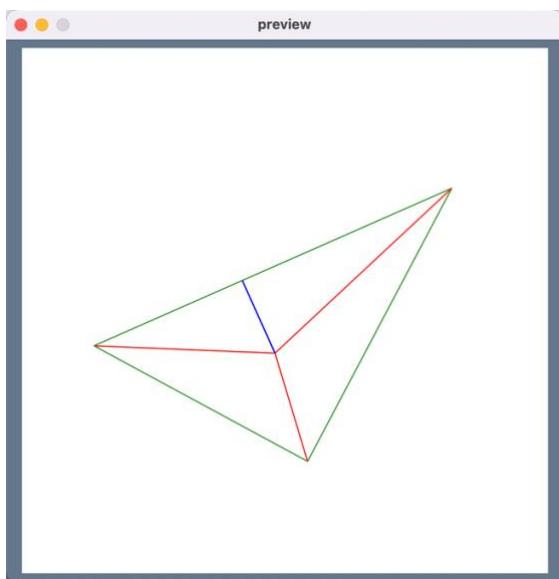


図 10 三角形の折り図（完成版）

凸四角形の実験結果を折り紙シミュレーターというサイト（折り図を入力すると、1~100%で折る様をシミュレートしてくれる）を使用して、正確な折り込みが行われたことを検証した（図 11、12、13）。だが、図 7 の正方形は凸四角形であるのに、正しく折られていない（四角形は形に問わず垂線は 2 本しか無い）。この事実と図 8 と三角形の結果を照らし合わせた結果、次の仮説が立てられる。ある多角形の全ての角の二等分線が一点で交わる（中心を持つ）場合、このプログラムは誤った結果を導く可能性がある。正 N 角形は中心を持つため、三角形と正方形、菱形だけを場合分けしても他の図形には応用出来ないため、三角形以外の図形においてある多角形が中心を持つ場合で場合分けを行う必要があると考えられる。



図 11 折り紙シミュレーターによる検証①

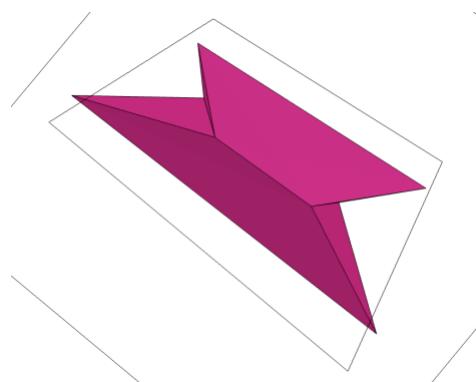


図 12 折り目 60% の状態

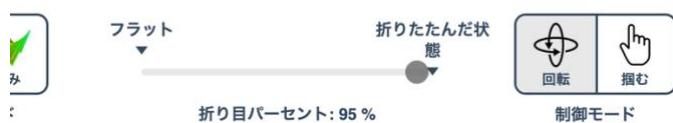
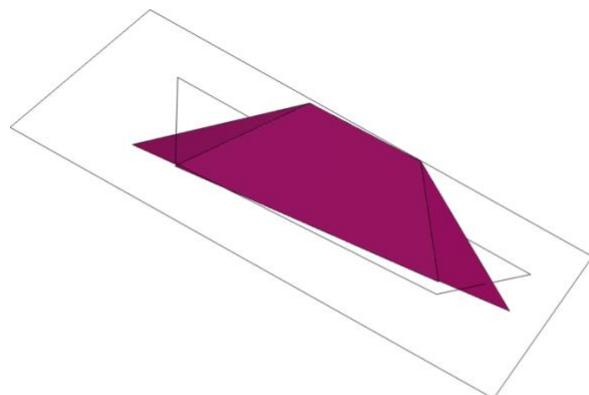


図 13 一直線になることの立証

凹四角形においては、複数の問題があると考えられるが、まずは角 A と角 B の二等分線が 2 回ぶつかっていることに違和感を覚えた。本プログラムは一つの二等分線に対して、隣り合う二つの角の二等分線の内、先に衝突する方を選んでいる。角 A にとって角 B は先だが、角 D には角 A が先なのである。また角 B と角 C は互いに一番近い。これらが全て引かれていたため、図 5 のような変な折り線が追加されてしまっていたのだと考えられる。この折り線追加を防ぐためには、次の条件を追加すれば良いと思われる。『①プログラムで初めて二等分線を衝突させる際は互いに持つ交点が同一である二角（図 5 では角 B と角 C）でなくてはならない。②既に交点がある二等分線はそれ以上延長しない。』①の目的はプログラムの最初の段階で誤った交点を求めないようにするためである。交点を選ぶプログラムは二等分線ごとに行われ、その順番は全二等分線が収容されたリストの 0 番目から行われる（間違った交点が先に求められる可能性がある）ため、順番を無視して、①に沿った交点の導出を行えば、正しい交点が最初に得られる。次の②は、二等分線が 2 回衝突するのを防いでいる。図 5 では、角 A と角 B の二等分線が繋がれているのが問題であるため、既に交点を持つ二等分線の延長を止めれば、2 回衝突するのは回避可能だ。①で交点の選定を行ったのは、先に正しい交点をえらばないと、2 回衝突するのを止めてても、間違った折り図であるのには変わりはないからだ。①、②は以上のことより、正しい交点を求める

ことが可能になる条件だと考えられる。

次に角 C に最も近い垂線が中途半端なところで止まっているのが、気になる点である(図6と同様のミス)。これは垂線を引く際のプログラムに沿っている体と考えられ、『ある2角(2頂点)A・Bの二等分線の交点から辺ABに向かって垂線を降ろす』というコードしか書かなかったため、こうなってしまったと考える。対処法としては“システム概要”で説明した 3β と 3γ の場合分けを行う必要があると考えられる。Pythonで角度を計算するにはベクトルの内積と大きさを使って計算をする。コードを組む過程で計算のエラーが幾度も発生してしまい、検証を繰り返して、修復した(図14はエラーの内容である)。試行錯誤を経て、角度の導出に成功したもの、ベクトルで算出したため、優角と劣角の区別が付かず、 3β や 3γ を実行するにはシステムとして不十分だと考えられる。アイディアとしては、『図形の内と外を区別し、角度を測ると同時に点などを置き、その点が図形の内側なら角度はそのまま、図形の外側なら 360° から測定した角度を引き、優角を求める』が考えられるが、本研究では実装はできないと考える。

これまでの成果は、折り図の出現と同時にターミナルに多角形内の角度が全て現れるようなプログラムにして記す(図15)。

```
(venv) akko9@MacBook-Air-2 OCSC % python3 app.py
{'cut': [
```

図14 エラー内容

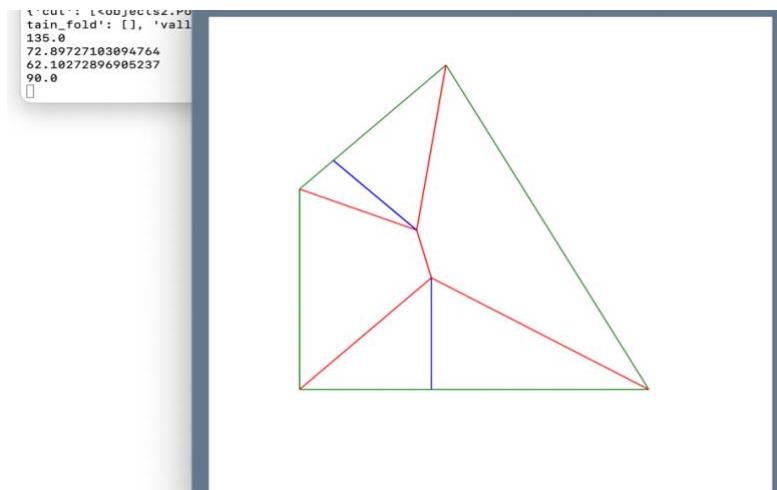


図 15 折り図と角度の算出

五角形の実験結果は予想通りではあった。本研究では交点を繋いだ後のプログラムまでは、書けなかったので、次回の研究への足掛かりへとしたい考察である。図 9 を見ると、凹四角形の最初の問題であった一つの二等分線が二つの交点を持ってしまっていることが確認できる。①、②（前考察に記述済み）により、交点を持つ二等分線の延長を不可能にした時、図内（五角形だと仮定）に残っているのは余った一つの角と二つの交点である（図 16）。この時点で折り紙を折ると、図 17 のように二つの角が消えて、立体の三角形ができていることが読み取れる。三角形の折り図は、全ての角の二等分線を引いて、内心から任意の一辺に垂線を降ろすというものである。これを立体の三角形でも行うことで一直線になると考えられる（立体の三角形でも内心があるというところまでは今回証明できていないが、複数の五角形の一刀切りを試行して観測できた再現性のある事象である）。だが、図 17 では二つの交点がある場所だけ隆起していて、錐体の頂点を成している。図 17 の立体図形を展開すると、図 18 のようになる。この図形を印刷して、昨年のように、手動で折り線を求める図 19 になる。（図中の GI と FJ は折った時に GE, FC とそれぞれ重なる線である）次に作りたいプログラムの内容は GH, FH を引くことである。図 20 は図 19 を折ったものであり、GH と FH は DG, BF の立体上の延長線であると見て取れる。以下は GH と FH の求め方である。

直線は平角定理より 180° であり、 $180^\circ = 360^\circ \div 2$ である。この式は直線が 360° の角を二等分していること（考え方であって、理論的に正しい訳ではない。 360° の角というのは角の定義に反しているため）を表している。直線の延長線ということは進んだ先の角度を二等分していなくてはならなく、図 17 より GH は $\angle DGD'$ 、FH は $\angle BFB'$ の二等分線だと考えられる。

$$\angle BFB' = 360^\circ - 2a \quad (\because \text{図 18}) \cdots ①$$

$$\angle DGD' = 360^\circ - 2b \quad (\because \text{図 18}) \cdots ②$$

$$(\text{ア}) \text{ より } \angle BFH = \angle BFJ + \angle CFH = 180^\circ - a = (360^\circ - 2a) \div 2 \cdots ③$$

$$(\text{イ}) \text{ より } \angle DGF = \angle DGI + \angle EGH = 180^\circ - b = (360^\circ - 2b) \div 2 \cdots ④$$

上記の式を計算するコードを組めば、五角形の折り図の作成に必要なコードは揃ったと言えるだろう。（a, b は角度を測る関数を使って、計算すれば良いと思われる）

六角形も同様のコードで動くと思われる。なぜなら、五角形は余る頂点（角）があるため、角が一つ増えても新たな交点が代わりに増え、変数を (a, b, c) に増やせば動くと思われる。（凹四角形のような例外があるため、作成時に改良が不可欠にはなるが、基本システムの指標は立てる事が出来たと思われる）

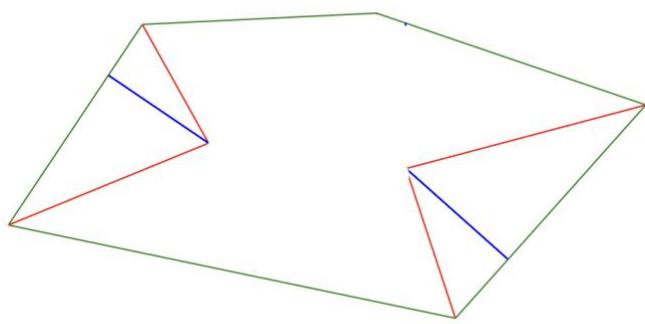


図 16 五角形の折り図 途中経過

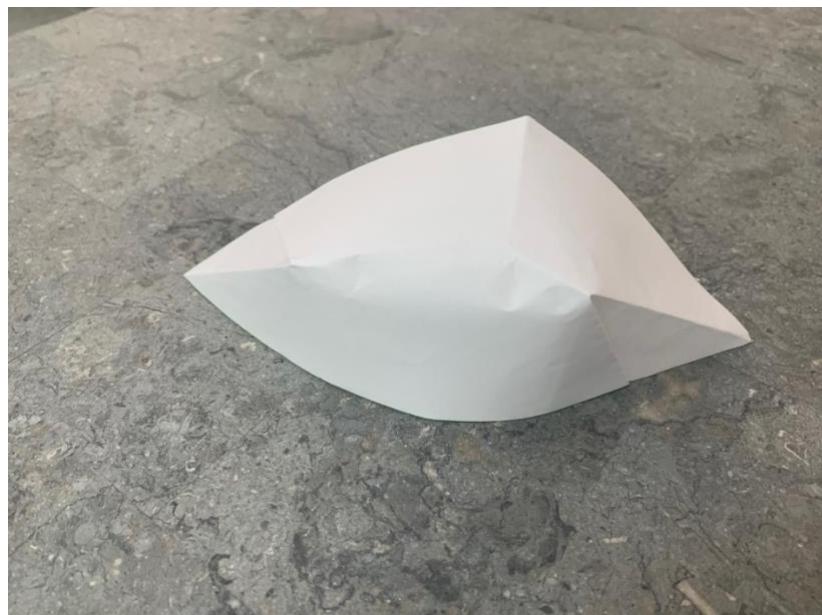


図 17 疋まれた五角形

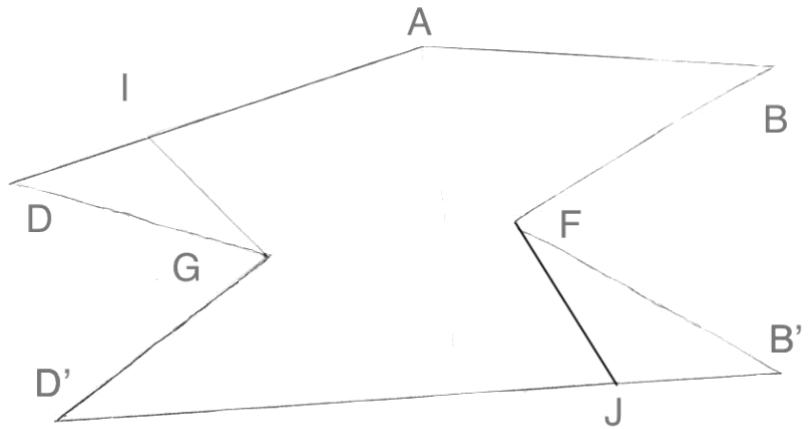


図 18 図 17 の展開図

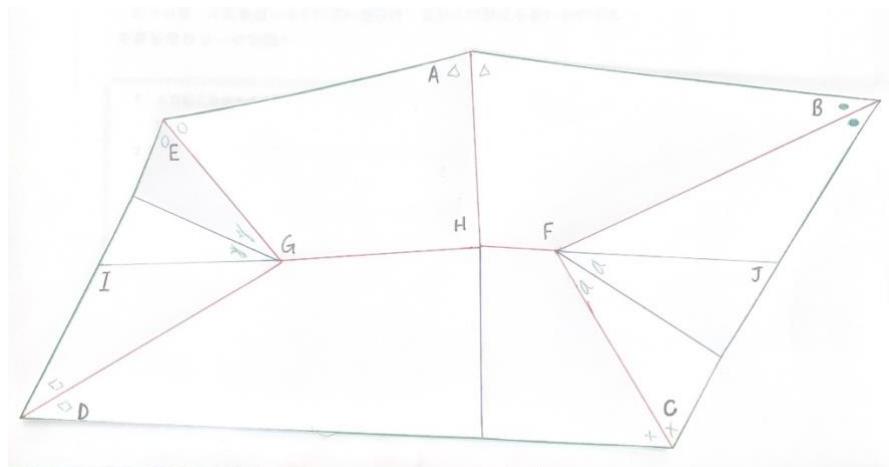


図 19 五角形の折り図

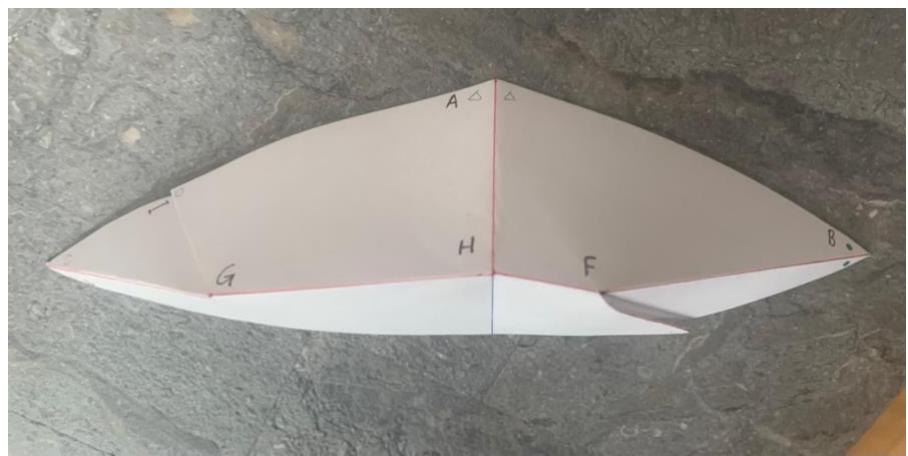


図 20 折り込んだ五角形

将来展望 :

考察で前述した通り、本研究では実装できなかったプログラムが多数あるので、それらの実装は今後の課題である。GUIについても、色々な機能の追加や、デコレーション等、改良できる点はある。本研究では平坦折り可能とする為の条件（偶数次数定理：頂点の次数は偶数である、前川・Justin 定理：各頂点のまわりで山折りと谷折りの本数の差は 2 である）はコード上に作成できなかったため、この概念を取り入れれば不要な折り線が追加されてしまう問題を解決する手段となりえるのではないかと考えている。

まとめ :

本研究では、一刀切りの折り図を作成するアプリケーションの開発をおこなったが、三角形と凸四角形（菱形・正方形は抜く）の完璧な作図が可能になったため、これらの図形においては正しいアルゴリズムの構築が出来た。研究を通して、一刀切り理論への理解が深まることの他に、今まで手動で作業していたことをコンピューターに自動化で行わせる難しさ、正しい結果に導くための指示の工夫など、新しい視点から得られた知見が沢山あった。今後世界は AI がプログラミングを行なっていく時代に入っていくけれど、何がプログラムされているのかを読み解いたり、自分の欲しいプログラムについて、論理的に抜けがないように、かつすっきりとした指示系統になるようにアルゴリズムが描けることはとても重要なスキルになると思う。これからもこのような観点からの発想を鍛えて、将来はイノベーションを起こす側の人となりたい。今年の労作展をきっかけに素晴らしい学びの機会を得ることが出来て、とても感謝しています。相談に乗ってくださった数学科の三浦先生、藤尾先生、Elnara 先生、Munetomo さん、Python の指導をして下さった小林さん、僕の研究を読んで下さった皆様、本当にありがとうございました。

付録 Appendix

Python プログラミング発展編 (Developed Programming) :

色 :

RGB(Red, Green, Blue)という色の表現方法があり、光の三原色である赤、緑、青を混ぜ合わせて膨大な数の色を表現する。それぞれの色の混ぜ具合（カラー値）に

0%~100%まであり、100%は 255 を表す。RGB では白は三原色をすべて 100%にした色、黒は何も混ぜていない色を表す。これらの色は白 = (255,255,255) [(100%,100%,100%)でも可]、黒 = (0,0,0) とプログラム上では表記する。なぜ最大値が 255 なのかというと、色一色あたりの情報量は 1byte であり、0~255 は 256 通りの値 (1byte=8bit=2⁸=256) を取るからである。

外部ライブラリ：

外部ライブラリは、元々自分のコンピュータに入っていないので、インストールの方法はコードの最初に「import ファイル名」と打てば使用可能である。だが外部ライブラリをインストールするには、ターミナルで「pip install ファイル名」と打たなければならない。

ライブラリというのは数学における定理のようなものである。事前に証明しておけば、問題で使えるが、証明していないと定理は使えず、問題の中で定理を説明しなければいけなくなるようなイメージに近い。例えば、任意の整数 x の三乗を知りたい時は、「math モジュール」をインストールすれば、 x に数を代入するだけで計算してくれる。しかし何もインストールしていない状態では、 $x^3=x*x*x$ と示さなければいけない。ライブラリを使うことによって、コードが簡略化され、読みやすく、綺麗なプログラムになる。

簡単な図形の描画：

一刀切りのアルゴリズムの自動化をする前に、Python プログラミングに慣れる必要があった。そこで扱ったコードが以下である。

1. 直線の描画

直線を描画するにあたって、まずは svgwrite というライブラリをファイルに追加（インポート）しなければならない。Svgwrite は SVG 画像を描画する上で大事な関数が沢山入っているパッケージである。そして svgwrite はキャンバスという図形や画像を描画するための領域を定義するためにも使われる。

直線の要素として必要なものは始点、終点、色である。これらの要素を直線としてキャンバスに描画していく。よって、青い (0,0) と(100,100)を結ぶ線のコードは図 21 のように作成可能である。

```

Practice > 🐍 svgwrite_test.py > ...
1   import svgwrite
2   canvas=svgwrite.Drawing("test.svg",profile="tiny")
3   canvas.add(
4       canvas.line(
5           start=(0,0),
6           end=(100,100,),
7           stroke=svgwrite.rgb(0,0,255,"%")
8       )
9   )
10  canvas.save()

```

図 21 line の獲得

2. 正方形の描画

直線が描けるなら、次段階として4本の直線を繋げれば、四角形ができる。尚、各頂点の座標を(a,b),(a,b+length x),(a+length x,b+length x),(a,b+length x)に設定すると(xは同値)正方形が作図可能である。シンプルに4本の線を別々のコードとして書くならば、図2のように書ける。

```

1   import svgwrite
2
3   canvas=svgwrite.Drawing("test.svg",profile="tiny")
4
5   canvas.add(
6       canvas.line(
7           start=(100,100),
8           end=(100,400,),
9           stroke=svgwrite.rgb(0,0,0,"%")
10      )
11  )
12
13  canvas.add(
14      canvas.line(
15          start=(100,400),
16          end=(400,400,),
17          stroke=svgwrite.rgb(0,0,0,"%")
18      )
19  )
20
21  canvas.add(
22      canvas.line(
23          start=(400,400),
24          end=(400,100,),
25          stroke=svgwrite.rgb(0,0,0,"%")
26      )
27  )
28  canvas.add(
29      canvas.line(
30          start=(400,100),
31          end=(100,100,),
32          stroke=svgwrite.rgb(0,0,0,"%")
33      )
34  )
35  canvas.save()

```

図 22 1辺 300 の黒色の枠の正方形

だが、このコードには重複している部分が多数あり、プログラミングでは、短くまとまっていた方が読む方も読みやすく、添削やミスの修正もしやすいので、コンパクトにまとめると、図3のようになる。図3では図1で作成した直線のコードを一般化したものに各4頂点の座標を代入することによって、図2を簡略化している。一般化は、*add_line()*という関数を作り、lineとして必要な要素は始点と終点と色であることを定義する（引き数の指定）。startという変数にはstart（始点）の座標を、endという変数にはend（終点）の座標を、strokeという変数にはr,g,b(%) (=colorであることは5行目で示した)を代入するという内容を8~10行で示した。そこに任意の数値群を4つ代入すること(14~17行)によってより簡単に正方形が作図できる。

```

1  import svgwrite
2
3  canvas=svgwrite.Drawing("origami.svg",profile="tiny")
4  def add_line(start,end,color):
5      r,g,b=color
6      canvas.add(
7          canvas.line(
8              start=start,
9              end=end,
10             stroke=svgwrite.rgb(r,g,b,"%"))
11         )
12     )
13
14 add_line((100,100),(400,100),(0,0,0))
15 add_line((100,100),(100,400),(0,0,0))
16 add_line((400,400),(400,100),(0,0,0))
17 add_line((400,400),(100,400),(0,0,0))
18 canvas.save()

```

図23 図22を簡略化したコード

3. 長方形の描画

図21~23までで、任意の直線の一般化をおこなったが、都度図23のように大量の直線を全座標付きでタイプするのは困難なので、長方形の一般化を行うことにした。

図3で作った*add_line()*という関数を使い、*add_rectangle()*という関数を作る。キャンバス、定点（始点）、幅、高さ、色の5要素を引き数に指定し、4本の直線を引いていく。図24ではx0,y0=startとし、1本目の線は(x0,y0)から(x0+幅,y0)への線、2本目の線は(x0,y0)から(x0,y0+高さ)への線、3本目、4本目も同様にx0,y0,幅,高さのみで表すことができる（具体的には図4を参照）。よって長方形は上記の5つの要素があれば描画できるといえる。最後に、この関数たちは、まとめて*add_shapes.py*と名付けられたファイルに保存した。

```

Practice > add_shapes.py > ...
1 import svgwrite
2
3 def add_line(canvas,start,end,color):
4     r,g,b=color
5     canvas.add(
6         canvas.line(
7             start=start,
8             end=end,
9             stroke=svgwrite.rgb(r,g,b,"%"))
10    )
11
12
13 def add_rectangle(canvas,start,width,height,color):
14     x0,y0=start
15     add_line(canvas,start,(x0+width,y0),color)
16     add_line(canvas,start,(x0,y0+height),color)
17     add_line(canvas,(x0+width,y0),(x0+width,y0+height),color)
18     add_line(canvas,(x0,y0+height),(x0+width,y0+height),color)

```

図 24 直線と長方形の一般化

この関数を使えば、図 5 のように即座に長方形が描ける。

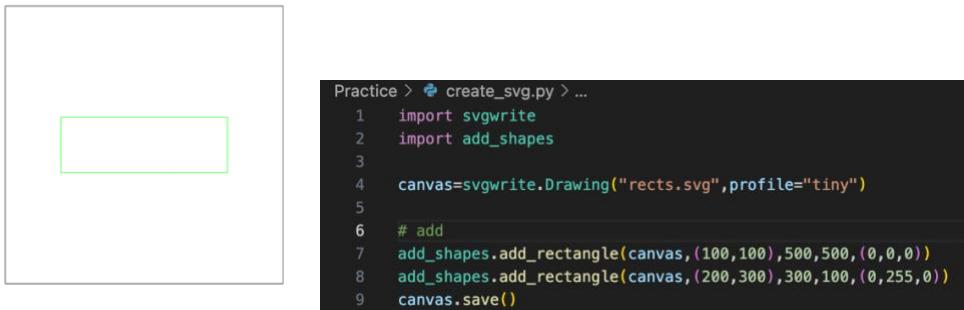


図 25 二つの長方形の描画

SVG 画像の読み取り :

SVG 画像は xml といった形式で扱われる。Xml 形式とは<>でタグ付けされているファイルであり、読み取りをする上で重要になってくるのは、図 26 の 6 行目以降の line の情報である。

```

1  <?xml version="1.0" encoding="utf-8" ?>
2
3  <svg baseProfile="tiny" height="100%" version="1.2" width="100%" xmlns="http://www.w3.org/2000/svg"
4  xmlns:ev="http://www.w3.org/2001/xml-events" xmlns:xlink="http://www.w3.org/1999/xlink">
5      <defs />
6      <line stroke="rgb(0%,0%,0%)" x1="100" x2="600" y1="100" y2="100" />
7      <line stroke="rgb(0%,0%,0%)" x1="100" x2="100" y1="100" y2="600" />
8      <line stroke="rgb(0%,0%,0%)" x1="600" x2="600" y1="100" y2="600" />
9      <line stroke="rgb(0%,0%,0%)" x1="100" x2="600" y1="600" y2="600" />
10     <line stroke="rgb(0%,100%,0%)" x1="200" x2="500" y1="300" y2="300" />
11     <line stroke="rgb(0%,100%,0%)" x1="200" x2="200" y1="300" y2="400" />
12     <line stroke="rgb(0%,100%,0%)" x1="500" x2="500" y1="300" y2="400" />
13     <line stroke="rgb(0%,100%,0%)" x1="200" x2="500" y1="400" y2="400" />
14  </svg>

```

図 26 SVG 画像のファイル

そして、この雑多な情報の中から、必要十分な要素だけを抜き取らないと、プログラムとして破綻してしまう。図 27 ではまず lxml というインストールした外部ライブラリの中の etree というモジュールをインポートした。この lxml というのは、xml 形式を解析、操作するためのライブラリであり、その中で etree は指示の親子関係を整理することができるモジュールである。

```

Practice > 🗂️ svg_parsing.py > ⚒ parse_svg
1   from lxml import etree
2
3  def parse_svg(svg_file_path):
4      with open(svg_file_path, 'rb') as file:
5          svg_content = file.read()
6          root = etree.fromstring(svg_content)
7          for element in root.iter():
8              yield element.tag.split("}")[ -1], dict(element.attrib)

```

図 27 SVG 画像を読み取る関数

次に *parse_svg ()* という関数を作っていく。

3 行目- *parse_svg ()* という関数を定義、引き数として SVG ファイル名 (svg_file_path) を受け取る。

4 行目- ファイルを読み取り (read 形式)、バイナリ (binary) 形式で開く。バイナリ形式では、データは 0 と 1 のバイナリ値の組み合わせで表現される。コンピュータは 0 と 1 の二つの数字しか使用しないと聞いたことはないだろうか？先に述べた三原色の種類も bit で表記されたものである。Bit は binary digit の略であり、二つとも 0 と 1 の 2 種類のみの数字で表記する。

5 行目- 中身を全て変数、svg_content に代入する。

6 行目- xml のツリー構造を作る。ツリー構造は、親要素と子要素 (xml 形式はタグ

と要素の組み合わせによってできている) の関係を持つ階層を形成する。例: 図 8

```
<?xml version="1.0"?>
<states>
    <state code="CA">
        <name>California</name>
        <capital>Sacramento</capital>
    </state>
    <state code="HI">
        <name>Hawaii</name>
        <capital>Honolulu</capital>
    </state>
    <state code="TX">
        <name>Texas</name>
```

図 28 ツリー構造の例

7 行目 - for 文というのは、中にあるプログラムをループさせることができると前述したが、今は含まれる要素を順番に取り出すことができる機能(順番に取り出すと言うのは、取り出すと言う行為をループさせたもの)の方が使われている。ここでは root(ツリー)に含まれる要素を順番に取得できる。

8 行目 - yield 文はデータを順番に返してくれる。element.tag.split("}")[−1]はまず要素のタグ情報(要素タグは一般に`{namespace}name`と表す)を `}` で分割し(`{namespace` と `name`)、後ろから 1 番目のデータ(`name`)を取得することができる。コンピューター上では数列は最初からだと (0,1,2,3...)、後ろからだと (-1,-2,-3...) とカウントする。次に dict(element.attrib) は詳細情報を辞書型で取得する。辞書型とは、キーとそれに対応する値(データ)をペアにして表示する型である。

*parse_svg()*という関数は任意の SVG ファイルに含まれる要素を辞書型で抜き取ることが可能である。

直線の獲得 :

*parse_svg()*を使い *get_lines.py* という直線の座標を導出するファイルを作る。
*parse_svg()*を使いたいので、*svg_parsing* から *parse_svg* をインポートする。
図 9 は色の識別式である。直線の要素は始点、終点、色だと前述したが、まずは色を RGB に変換して、実数値を導出するプログラムの作成をおこなった。

色の獲得

*color_converter()*という関数の引き数として *stroke*(色を表す文字列)を受け取るようにする。*rgb* という空箱を予め作っておき、ここにデータを収納する。5 行目では *rates*=文字列`stroke`の 4 番目から最後から 1 番目の文字までを部分文字

列とし、取得した文字列（例：{'stroke': 'rgb(0%,0%,0%)', 'x1': '100', 'x2': '600', 'y1': '100', 'y2': '100'}という情報（辞書型）にこの操作を行うと、「rgb(0%,0%,0%)」のみを取得できる。）を `split(",")`（カンマ（,）で分割し、各成分を格納する）したデータ群。

For文を使い、`rates` というデータ群から、`rate` を順番に持ってくる。そして、`rgb.append(int(rate[:-1])*255//100)` は for 文の中に含まれるため、`rate` は `rgb` という箱に `append`（収納）される。間のプロセスとしては、`split` で分割したデータの-1番目（例の値では 0% と 0% と 0% が求められる）を `int` というプログラムで整数に変換し、255 倍して 100 で割ることにより、% を整数のカラー値に変換することが可能になる（この//は切り捨て除算を表す記号で、商の少数点以下を切り捨てる）。なぜ%のままではダメなのかというと、辞書型のデータは文字列であるため、コンピュータは数字ではなく文字として認識してしまうからである。最後に求めた値を `rgb` という箱に `return`（入れる、返す）すれば、色のカラー値を求める関数が出来上がる。

```
1   from svg_parsing import parse_svg
2
3   def color_converter(stroke):
4       rgb = []
5       rates = stroke[4:-1].split(",")
6       for rate in rates:
7           rgb.append(int(rate[:-1])*255//100)
8
9   return rgb
```

図 29 色を識別する関数

頂点の獲得

色の次は頂点である。まずは for 文で `tag`（タグ）と `attribute`（オブジェクトが持つデータ）を `parse_svg()` から順番に受け取ることにする。次に if 文で、もしタグが `line` ならば 14~16 行目の操作を行うと書く。下記のコードが実際の線の中身である。

```
<line stroke="rgb (0%,0%,0%)" x1="100" x2="600" y1="100" y2="100" />
```

線には `line` というタグがついているため、線かどうかはタグを調べれば識別可能である。

14 行目 - `Color` には `color_converter()` で得られた `stroke` のデータを返している。
15,16 行目 - `vertex_1,2` にはそれぞれ x 座標である `x1,x2` と y 座標である `y1,y2` を `float` した（浮動小数点数にした）値を代入しろという命令である（浮動小数点数とは整数と違って、小数点以下の桁数の表現ができ、さらに、精度の高い数が表現可能である。また数学的計算にも適している）。

最後に `print(color,vertex_1,vertex_2)` により、線の要素として必要な 3 要素を文字列では無く数字として受け取ることができるようになった（文字列は情報として扱いに

くい)。これにより任意のファイルの中にあるすべての線の情報が扱いやすい形式で獲得可能になった。

```
11  for tag,attribute in parse_svg(svg_file_path="rects.svg"):
12      if tag=="line":
13          # print(attribute)
14          color = color_converter(attribute['stroke'])
15          vertex_1 = (float(attribute['x1']),float(attribute['y1']))
16          vertex_2 = (float(attribute['x2']),float(attribute['y2']))
17
18          print(color,vertex_1,vertex_2)
```

図 30 直線の情報の取得

```
Practice > ✎ get_lines.py > ...
1  from svg_parsing import parse_svg
2
3  def color_converter(stroke):
4      rgb = []
5      rates = stroke[4:-1].split(",")
6      for rate in rates:
7          rgb.append(int(rate[:-1])*255//100)
8      return rgb
9
10
11 for tag,attribute in parse_svg(svg_file_path="rects.svg"):
12     if tag=="line":
13         # print(attribute)
14         color = color_converter(attribute['stroke'])
15         vertex_1 = (float(attribute['x1']),float(attribute['y1']))
16         vertex_2 = (float(attribute['x2']),float(attribute['y2']))
17
18         print(color,vertex_1,vertex_2)
19
```

図 31 モジュール get_lines.py

コードの説明 (Code Explanation) :

(本編の工場の番号通り記載、だが、工場②では①で作った関数を大量に使用しているため、工場②から解説していく)

工場② : **objects.py** : 図形の鋳型(クラス)を定義/保管している工場(図書館?)

i. クラス Point を定義

Import NumPy as np は numpy という計算に優れたライブラリをインポートして、これからは np と略して使いますよという意味である。

まずは`__init__()` (関数) が引き数に position を受け取り、それを使用して点の

座標を初期化している。position は要素として x 座標 y 座標を持っている。それらの座標を self.x と self.y に保存している。さらに self.position=position から position の座標は self.position にも保存されている。この関数により、この点（求めたい点）の x,y 座標が求められる。

次に *get_position_vector()* (関数) が self.position を np.array() (Python の標準リストを numpy の多次元配列に置き換えることができる関数) で数値データを効率的に操作するための機能や関数を利用することができる配列にする。要約すると、文字列を数字（位置・座標ベクトル、又は点座標）に置き換えた。

最後に *distance_to()* (関数) では引き数としてこの点と、別の点を受け取っている。その時、self.get_position_vector() (この点の座標ベクトル) から another_point.get_position_vector() (別の点の座標ベクトル) を引くことによって二点間のベクトルの長さを np.linalg.norm() (数学的計算を行う関数) という関数を用いて求めている。

これを組み合わせて、作ったコードは、2つの点の座標を使用して点を初期化し、その点と別の点との間の距離を計算するためのクラスの定義をしている。

```

1 import numpy as np
2
3 class Point():
4     def __init__(self, position):
5         self.x, self.y = position
6         self.position = position
7
8     def get_position_vector(self):
9         return np.array(self.position)
10
11     def distance_to(self, another_point):
12         return np.linalg.norm(self.get_position_vector() - another_point.get_position_vector())
13

```

図 32 class Point

ii. クラス Segment を定義

先と同様に *__init__()* という関数を最初に入れる。引き数として vertices(頂点), start(始点)を受け取っている。start=None は start のデフォルト値が“存在しない値”であることを表す。self.v1 と self.v2 は一つ目の頂点と二つ目の頂点を表すオブジェクトであり、=map(Point, vertices) は vertices という頂点情報をオブジェクトに変換している（データの置換を行わないと、オブジェクトに座標を入れることが出来ない）。if 文を使い、not start ==None (start が None ではない=start に値がある) である時、self.start = Point(start) (Segment の始点は Point の始点と一緒に点) である。次の else 文の意味は、Segment に start がない時は self.start = self.v1 (Segment の始点は一つ目の頂点) であることを言っている。最後に、vertices を self.vertices に保存する（この行は else 文に含まれない）。

元々、この関数には start という引数は入っておらず、if, else で場合分けする必要もなかった。だが、後になって二等分線の交点を求める際に、線分の始点がわかつていないと、ある頂点が隣り合う二つの頂点の内、どっちの二等分線と交わらせたらいいのかが判別できない（判別は、ある頂点から交点までの距離が短い方が選ばれる）ため、この関数に追加することになった。

追加で、self.direction_vector の定義も行っている。Segment のベクトルは始点と終点のベクトルの差で表現される。次の行では、ベクトルを自身のベクトルの長さで割ることにより、単位ベクトル（長さ 1 のベクトル）を作成、獲得している（プログラムの計算でエラーが発生するのを防ぐ）。

```

16     def __init__(self, vertices, start=None):
17         self.v1, self.v2= map(Point, vertices)
18         if not start == None:
19             self.start = Point(start)
20         else:
21             self.start = self.v1
22         self.vertices = vertices
23
24         self.direction_vector = self.v2.get_position_vector()-self.v1.get_position_vector()
25         self.direction_vector/= np.linalg.norm(self.direction_vector)

```

図 33 start の場合分け

線分の式を求めるには、2 頂点の座標が必要であり、それぞれ self.v1 と self.v2 と決めたので、求めることが可能である（関数：get_equation()）。

（計算のプロセスは図 14 を参照）

```

29     def get_equation(self):
30         a = self.v2.y - self.v1.y
31         b = self.v1.x - self.v2.x
32         c = (self.v2.x * self.v1.y) - (self.v1.x * self.v2.y)
33         return a, b, c

```

図 34 直線の一般形を求める式

次に線分の交点を求めていく。まずは、2 直線を定義する。直線の一般形は $ax+by+c=0$ であり、使いたいのは a, b, c（係数 + 定数項）なので、2 つの直線の定義に必要なのは、それぞれの直線における a, b, c に値する部分である。また 3 つの数は get_equation() で、各直線の 2 頂点から求められる。

```

def get_intersection(self, another_segment):
    a1, b1, c1 = self.get_equation()
    a2, b2, c2 = another_segment.get_equation()

```

図 35 二直線の一般形の定義

線分の交点を求める目的は、二等分線の交点が後々必要になるからである。2直線の交点を求める時、連立方程式を解けば、解が求められるが、逆行列の概念（後述する）を使いたいので、行列方程式を使って解いていく。行列とは、一般に図16のように表すことができ、複数のベクトルを行（横ベクトル）と列（縦ベクトル）に並べて構成された2次元の数表である。このプログラムの構築に必要なのは、行列の考え方のみで、数学的計算は `np.linalg.solve` という関数が行うため立式をするだけでよい。

図17の39行目、`A = np.array([a1,b1],[a2,b2])` は、 2×2 の行列 A を作っている。そして44行目、`b = np.array([-c1,-c2])` は`b`というベクトルを作っている。
これは次の式をベクトルに置きえたものである

$$\begin{aligned} a_1x + b_1y &= -c_1 \\ a_2x + b_2y &= -c_2 \end{aligned}$$

具体例：

$$\begin{aligned} x + y &= 9 & (1) \\ 2x + 4y &= 22 & (2) \end{aligned}$$

図36 連立方程式の例

(1) (2)は次の行列に変換可

$$\begin{pmatrix} 1 & 1 \\ 2 & 4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 9 \\ 22 \end{pmatrix} \quad (3)$$

図37 (1) (2)の行列化

$$\left(\begin{array}{cc|c} 1 & 1 & 9 \\ 2 & 4 & 22 \end{array} \right) \quad (4)$$

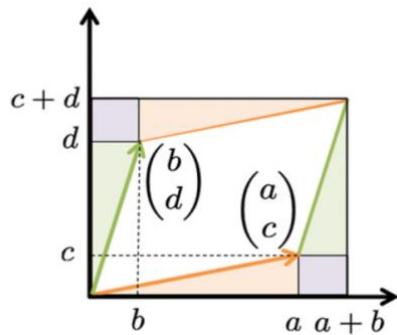
図38 (3)の簡素化

(4)は一般化すると上記の $Ax = b$ というふうに置き換えることが可能である。そして、これをコンピューターに計算させると、解の導出ができる。ただし、2直線が平行の時、2直線は交点を持たない。よって、2直線が平行かそうで無いかを場合分けしなければならない。

40~42行目では、まず determinant = A だと定義した。次に、if文で determinant の値が 0 に近似している場合、False を返してと命令する。上記の A の行列式は 2 直線が平行でない時、非ゼロの値になる。逆に 2 直線が平行の時、行列式は 0 となる。なぜなら、行列 A は行列式に変換すると、 $a_1b_2-a_2b_1$ になり、2 直線が平行の時、それらの直線は同じ方向を示し（傾きが同じ）、 $a_1b_2-a_2b_1=0$ となるからだ。これにより、交点が存在するならば、その座標が求められるようになった。演算は 45 行目で行い、座標は intersection_point として保存する。

先述した通り、頂点から交点までの距離を比較する必要があるので、前作った *distance_to()* 関数を使って距離を測定する。最後に return で distance と intersection_point を返して、交点を求める関数は完成する。

行列 $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$



行列式 $|A| = ad - bc$

図 39 行列の例

```

35     def get_intersection(self, another_segment):
36         a1, b1, c1 = self.get_equation()
37         a2, b2, c2 = another_segment.get_equation()
38
39         A = np.array([[a1,b1], [a2, b2]])
40         determinant = np.linalg.det(A)
41         if np.isclose(determinant, 0):
42             return False, False
43
44         b = np.array([-c1, -c2])
45         x, y = np.linalg.solve(A, b)
46
47         intersection_point = Point((x, y))
48
49         distance = self.start.distance_to(intersection_point)
50
51         return intersection_point, distance

```

図 40 2 直線の交点を求める関数

では、任意の点から Segment に向かって垂線を降ろす関数を作る。この時、引
き数には point_obj(点)と self(Segment)が必要になる。`__init__()`で作った
`self.direction_vector(Segment の単位ベクトル)`を `d` として扱い、`np.array([-
d[1],d[0]])`が Segment に垂直なベクトルを作っている。垂直が生まれる訳として
は、`[-d[1],d[0]]`が本来のベクトルの x 成分を y 成分として受け取り (`d[0]` は x 座
標)、y 成分は-1 倍して x 成分に受け取らせているからだ (一次関数において、傾き
a の直線があったとき、これに垂直な線の傾きは $-1/a$ である。これは元の線が x 方向
に 1 進んだ時に、y 方向に a 進むことを表し、垂線は x 方向に $-a$ 進めば、y 方向に 1
進むことを表す。ベクトルはこの x,y 方向に z 進むことを数値化したものである)。

`p1, p2` は垂線 (単位ベクトル) の始点、終点であり、このベクトルを
`unit_perpendicular` とする。次の文では、この単位ベクトルと Segment の交点 (垂線
の足) を求めている。最後に `return` で、`p1` を始点、求めた垂線の足を終点とする新
たな Segment を返している。

```
53     def get_perpendicular(self, point_obj):
54         d=self.direction_vector
55         perpendicular_d=np.array([-d[1],d[0]])
56
57         p1 = point_obj.get_position_vector()
58         p2 = p1+perpendicular_d
59
60         unit_perpendicular= Segment([p1,p2])
61         intersection_point, _ =self.get_intersection(unit_perpendicular)
62
63         return Segment(p1.tolist(), intersection_point.position)
```

図 41 垂線を任意の点から降ろす関数

```

15  class Segment():
16      def __init__(self, vertices, start=None):
17          self.v1, self.v2= map(Point, vertices)
18          if not start == None:
19              self.start = Point(start)
20          else:
21              self.start = self.v1
22          self.vertices = vertices
23
24          self.direction_vector = self.v2.get_position_vector()-self.v1.get_position_vector()
25          self.direction_vector/= np.linalg.norm(self.direction_vector)
26
27
28
29      def get_equation(self):
30          a = self.v2.y - self.v1.y
31          b = self.v1.x - self.v2.x
32          c = (self.v2.x * self.v1.y) - (self.v1.x * self.v2.y)
33          return a, b, c
34
35      def get_intersection(self, another_segment):
36          a1, b1, c1 = self.get_equation()
37          a2, b2, c2 = another_segment.get_equation()
38
39          A = np.array([[a1,b1], [a2, b2]])
40          determinant = np.linalg.det(A)
41          if np.isclose(determinant, 0):
42              return False, False
43
44          b = np.array([-c1, -c2])
45          x, y = np.linalg.solve(A, b)
46
47          intersection_point = Point((x, y))
48
49          distance = self.start.distance_to(intersection_point)
50
51          return intersection_point, distance
52
53  def get_perpendicular(self, point_obj):
54      d=self.direction_vector
55      perpedicular_d=np.array([-d[1],d[0]])
56
57      p1 = point_obj.get_position_vector()
58      p2 = p1+perpedicular_d
59
60      unit_perpendicular= Segment([p1,p2])
61      intersection_point, _ =self.get_intersection(unit_perpendicular)
62
63      return Segment([p1.tolist(), intersection_point.position])

```

図 42 class Segment

iii. クラス Polygon を定義

今までと同じくクラスには初めに `__init__()`を入れるが、今回は多角形のクラスが作りたいので、頂点さえあれば図形は描画できる。よって、引き数としては `vertices`だけを受け取れば良い。

```
65     class Polygon():
66         def __init__(self, vertices):
67             self.vertices = vertices
```

図 43 `vertices` の初期化

多角形の構成要素は辺と頂点である。頂点は自分で入力するので、入力した頂点を繋いで、辺にする必要がある。図 44 では、まず `segments` という空箱を作っている。次に for 文で `i, v2` を `enumerate` (リスト内で要素とそれに対応するインデックス(リストの何番目なのかを表す)を同時に返せる) している。`i` は自分の頂点、`v2`には次の頂点 (リストでは頂点は時計回りに保存されている) が代入される。次に `v1` には `self.vertices` というリストの `i` の 1つ前の頂点であると言っている。これによりある点 `i` に隣接する 2 頂点 `v1, v2` が得られる。最後に `i` と `v1`、`i` と `v2` で作られる `Segment`(辺)を最初に作った `segments`(空箱)に `append`(収納)すれば全ての辺が獲得可能になる (for 文はループさせるため、`i` には全ての頂点が代入される)。

```
69         def get_segments(self):
70             segments = []
71             for i, v2 in enumerate(self.vertices):
72                 v1 = self.vertices[i-1]
73                 segments.append((v1, v2))
74             return segments
```

図 44 多角形を構成する辺を取得する関数

ここから漸く直線骨格の作成に入る。まずは角の二等分線を引くコードの作成である。77~79 行目では、二等分線を引く角に注目している。`target_vertex` が二等分線を引く角、`prev_vertex` と `next_vertex` は前後の隣接する頂点である。

`target_vertex` の `self.vertices`(リスト)における `vertex_index`(位置情報)は `target` の点と一緒にである (78 行目)。`prev_vertex` の位置情報は `vertex_index-1`(`target_vertex` の一つ前の点)である (77 行目)。だが、`next_vertex` は一捻りしなければならない。単純に `vertex_index+1` で入力するとプログラムはエラーを吐いてしまう。リストにおける一つ前は-1 で表現可能である。なぜなら、リストは後ろから-1,-2,-3…と続いているため、引き算は可能である[例：リストの二番目から 3 つ戻りたいときは `2-3=-1` と行

き着く先は一番後ろのインデックスである]。だが、足し算には注意が必要となる。プラス操作を行うには “ $+1 \% \text{len}(\text{self.vertices})$ ” ($\text{vertex_index}+1$ という数を `self.vertices` の長さ（要素の数）で割ったあまりを求めている) を付けることで足し算を可能にしたのである[例：要素が 5 つのリストの 5 番目から 2 つ進みたい時は $(5+2)\%5=2$ で 2 番目と求められる] (※ % は割ったあまりを求められる)。

```
77     prev_vertex = Point(self.vertices[vertex_index-1])
78     target_vertex = Point(self.vertices[vertex_index])
79     next_vertex = Point(self.vertices[(vertex_index+1)%len(self.vertices)])
```

図 45 二等分線を弾くために必要な頂点の定義

66~67 行目ではベクトルの値を定めている。`vector1` は `prev_vertex` から `target_vertex` へのベクトルを、`Vector2` は `next_vertex` から `target_vertex` へのベクトルを求めている。

```
66     vector1 = prev_vertex.get_position_vector() - target_vertex.get_position_vector()
67     vector2 = next_vertex.get_position_vector() - target_vertex.get_position_vector()
```

図 46 `target_vertex` へのベクトル

二等分線は単位ベクトル（長さ 1 のベクトル）の足し算によって可能である。

以下の図 47 がロジックである。

| ひし形の対角線が内角を二等分することを利用する。

\vec{a}, \vec{b} と同じ向きの単位ベクトルは $\frac{\vec{a}}{|\vec{a}|}, \frac{\vec{b}}{|\vec{b}|}$ である。

ここで、 $\overrightarrow{OA'} = \frac{\vec{a}}{|\vec{a}|}$, $\overrightarrow{OB'} = \frac{\vec{b}}{|\vec{b}|}$, $\overrightarrow{OD} = \frac{\vec{a}}{|\vec{a}|} + \frac{\vec{b}}{|\vec{b}|}$ とする。

このとき、 $OA'DB'$ はひし形となり、直線 OD が $\angle AOB$ を二等分する。

また、点 P は直線 OD 上にあるから、 $\overrightarrow{OP} = t\overrightarrow{OD}$ が成り立つ。

$$\therefore \vec{p} = t \left(\frac{\vec{a}}{|\vec{a}|} + \frac{\vec{b}}{|\vec{b}|} \right)$$

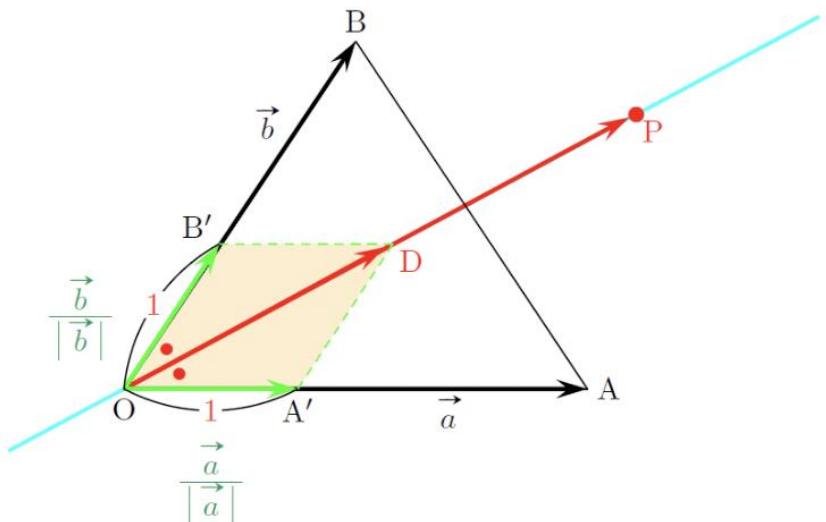


図 47 ベクトルを使った二等分線の引き方

よって、取得したベクトルを自身のベクトルの長さで割ってあげることにより、単位ベクトルが手に入る (`unit_vector1,2`)。(`np.linalg.norm` は長さを計算する関数) そして図より `unit_vector1` と `unit_vector2` を足し合わせて、`bisector_vector` という角を二等分するベクトルができた。先と同様の方法で、単位ベクトルの獲得もしている。

```

unit_vector1 = vector1 / np.linalg.norm(vector1)
unit_vector2 = vector2 / np.linalg.norm(vector2)

bisector_vector = unit_vector1 + unit_vector2
bisector_unit_vector = bisector_vector / np.linalg.norm(bisector_vector)

```

図 48 単位ベクトルと角を二等分するベクトルの取得

この関数は引き数に `in_point`, `out_point` というデフォルト値=False というものを持っている。最後の条件分岐 (if-elif-else) 文は二等分線の始点、終点 (`in_point`, `out_point`) が存在するかしないかで場合分けを行っており、詳しくは説明しないが、二等分線のベクトルを任意の値 (もう一つの引き数、`length=1` の数字を変更することによって、長さの調節可) で求めることを可能にした。

```

76     def get_bisector(self, vertex_index, length=1, in_point=False, out_point=False):
77         prev_vertex = Point(self.vertices[vertex_index-1])
78         target_vertex = Point(self.vertices[vertex_index])
79         next_vertex = Point(self.vertices[(vertex_index+1)%len(self.vertices)])
80
81         vector1 = target_vertex.get_position_vector() - prev_vertex.get_position_vector()
82         vector2 = target_vertex.get_position_vector() - next_vertex.get_position_vector()
83
84         unit_vector1 = vector1 / np.linalg.norm(vector1)
85         unit_vector2 = vector2 / np.linalg.norm(vector2)
86
87         bisector_vector = unit_vector1 + unit_vector2
88         bisector_unit_vector = bisector_vector / np.linalg.norm(bisector_vector)
89
90         if in_point and out_point:
91             bisector_obj = Segment((in_point, out_point), start=target_vertex.position)
92         elif in_point:
93             bisector_obj = Segment((target_vertex.position, in_point))
94         elif out_point:
95             bisector_obj = Segment((target_vertex.position, out_point))
96         else:
97             out_point = list(target_vertex.get_position_vector() - bisector_unit_vector* length)
98             in_point = list(target_vertex.get_position_vector() + bisector_unit_vector* length)
99             bisector_obj = Segment((in_point,out_point),start=target_vertex.position)
100
101    return bisector_obj

```

図 49 角の二等分線を求める関数

頂点の角度を計算する関数が考案にて、必要になったので、この class `Polygon` に挿入した。角度の獲得はベクトルの内積と大きさを使って算出するため、二等分線を求めた時と同様に、`vector1, vector2` を作る (角度の求め方には公式が存在するため、本実験ではそれを使用する)。次にこのベクトルの x 成分と y 成分を取り出す (111~114 行目)。次に、それぞれのベクトルの長さを測定するのにはピタゴラスの定理を使う。x 成分と y 成分とベクトルの長さを三角形の底辺、高さ、斜辺に見立て、計算する。ルートを表す記号はないので、`math.sqrt()` という関数を使用する。

次にベクトルの内積を求める。A[x1, y1], B[x2, y2]という二つのベクトルがあった時に、内積は $x_1 \times x_2 + y_1 \times y_2$ で求めることができる。そして、角度を求めるにあたり、次の公式が存在する。

$$\cos \theta = \text{内積} \div (\text{vector1 の長さ} \times \text{vector2 の長さ})$$

θ とは求めたい角度であり、 \cos は三角関数の一つである（重要なのは関数であるということ）。関数なので、上記の式は「 $\theta =$ 」の形に変換することが可能であり、 \arccos という逆三角関数を使うことで角度を求めることが可能になる（ \arccos （定数）は一つの解を持つ）。また「 $\theta =$ 」に書き換えた式は以下の通りである。

$$\theta = \arccos(\text{内積} \div (\text{vector1 の長さ} \times \text{vector2 の長さ}))$$

この式に含まれる変数は全て、求められるので、 θ は計算可能である。ここで得られる角度の単位はラジアン（ $180^\circ = \pi$ とする三角関数などで使われる単位

例： $90^\circ = \pi/2$ であるため、これを角度に直すには `math.degrees()` という関数に計算してもらう必要がある（同様に `arccos` の計算も `acos()` という関数に計算をしてもらう）。これにより、角度を求めることが可能になった。得られた角度の確認がしたいので、`print(angle)` でターミナルに角度を表示可能にする。

```

103     def get_angle(self, vertex_index):
104         prev_vertex = Point(self.vertices[vertex_index-1])
105         target_vertex = Point(self.vertices[vertex_index])
106         next_vertex = Point(self.vertices[(vertex_index+1)%len(self.vertices)])
107
108         vector1 = target_vertex.get_position_vector() - prev_vertex.get_position_vector()
109         vector2 = target_vertex.get_position_vector() - next_vertex.get_position_vector()
110
111         x_component_vector1 = vector1[0]
112         y_component_vector1 = vector1[1]
113         x_component_vector2 = vector2[0]
114         y_component_vector2 = vector2[1]
115
116         scale_vector1 = math.sqrt(x_component_vector1**2 + y_component_vector1**2)
117         scale_vector2 = math.sqrt(x_component_vector2**2 + y_component_vector2**2)
118
119         inner_product_vector = x_component_vector1*x_component_vector2 + y_component_vector1*y_component_vector2
120
121         angle = math.degrees(math.acos(inner_product_vector / (scale_vector1 * scale_vector2)))
122         print(angle)

```

図 50 角度を求める関数

図 49 で作成した二等分線を求める関数だけではどこで他角の二等分線と交差させたら良いかが不明瞭である。よって、これから作るのはどの角の二等分線同士を交差させれば良いのかを識別できる関数である。

この関数の引数は `self` のみである。なぜならある図形の直線骨格が欲しいのに、図形以外の情報が必要なプログラムは不便であるからだ。この関数では二等分線やそれらの交点が非常に沢山生成されるため、空箱を作りリストにしないと、情報が交錯してしまう。最初に、`edges` という多角形の辺もリストにする（`map` 関数で取り出した要素を `list` 関数でリストにする）。次に `bisectors`, `intersections`, `perpendiculars`, `angles` と言う空箱をまず作る。（図 51）

```

124     def get_bisectors(self):
125         edges = list(map(Segment, self.get_segments()))
126         perpendiculars = []
127         bisectors = []
128         intersections = []
129         angles = []

```

図 51 リストと空箱作成

次の 2 行では、self.vertices の length(len) (要するに、頂点の数) 中、range(0~1 の範囲の整数を生み出す)で各頂点のインデックスを取得し、vertex_index とリンク (in) させることで、vertex_index から頂点の座標などの情報にアクセスできるようになる。そして、得られた頂点座標から *get_bisector()* で角の二等分線(ベクトル形式)を求め、それを bisectors.append でリストに格納する。この手順を for 文でループさせているため、全てのベクトルを取得できる。同様に、角度を *get_angle()* で求め、angles に格納した。

```

for vertex_index in range(len(self.vertices)):
    bisectors.append(self.get_bisector(vertex_index))

for vertex_index in range(len(self.vertices)):
    angles.append(self.get_angle(vertex_index))

```

図 52 二等分線、角度の獲得

次の図 53 では、まず prev_bisector と next_bisector の定義を行っている。その前に、target_bisector=i とし、これを bisectors (リスト) に enumerate している。この 1 文が bisectors (リスト) の位置を指定するだけで要素 (始点、長さ等の情報) を全て引っ張ってこられるのを可能にしている (enumerate は要素と位置(index)をペアにして生成可能)。

138,139 行目の内容は prev_vertex と next_vertex の定義(*get_bisector* に記載)と殆ど同じなので、説明は省略する。141,142 行目で *get_intersection()* の return で返される変数が prev_intersection, prev_distance, next_intersection, next_distance と形式が一緒なので、引数に prev, next_bisector を代入することによって求められる。

```

137     for i, target_bisector in enumerate(bisectors):
138         prev_bisector = bisectors[i-1]
139         next_bisector = bisectors[(i+1)%len(bisectors)]
140
141         prev_intersection, prev_distance = target_bisector.get_intersection(prev_bisector)
142         next_intersection, next_distance = target_bisector.get_intersection(next_bisector)

```

図 53 交点と距離の獲得

図 54・55 は図 53 の for 文の中に含まれる。その中に if 文でもし隣接する二つの頂点のなす角の二等分線と target_bisector が交点を持つ時、次のふた通りに場合分けできる。

[① もし prev_distance > next_distance である時、リスト : bisectors の i 番目 (self) には距離が短い next_intersection と target_vertex が結ばれる。そして、リスト : intersections に next_intersection の座標を *append* している。

また、next_intersection から i の次の辺に向かって垂線を下ろしている。i の次の辺はプログラム上 next_vertex と target_vertex を 2 頂点とする辺であるため

(get_segment() の始点、終点である v1,v2 は vertices の i, i-1 である =edges の i は target_vertex と prev_vertex を結んだ辺である)、降ろしたい辺に向かって、引ける垂線を空箱に *append* できる]

[② else 文は①(if 文)の余事象である。これは prev_distance < next_distance であることを示していて、①と同様に距離が短い prev_intersection が交点に選ばれ、その座標がリストに *append* されている。垂線も①とほとんど同じプログラムで引くが、垂線を降ろす辺を edges の i に変更して適切な垂線を perpendiculars に *append* できる]

```

150     if prev_intersection and next_intersection:
151         if prev_distance >= next_distance:
152             bisectors[i] = self.get_bisector(i, in_point=next_intersection.position)
153             intersections.append(next_intersection.position)
154             perpendiculars.append(edges[(i+1)%len(edges)].get_perpendicular(next_intersection))
155         else:
156             bisectors[i] = self.get_bisector(i, in_point=prev_intersection.position)
157             intersections.append(prev_intersection.position)
158             perpendiculars.append(edges[i].get_perpendicular(prev_intersection))

```

図 54 1 つの角の二等分線が隣接する 2 つの角の二等分線と交点を持つ場合

図 55 では elif 文(else if : 意味は 2 番目以降の if、条件が 2 つ以上ある時 (else は条件を全て満たさないので機能しない) のみ使用可)で、prev_intersection のみ存在する場合、図 33 の条件文②と同様に、prev_intersection の座標で、intersections に *append* され、垂線も同様に perpendiculars に *append* されている。また、next_intersection のみ存在する場合は①のようになる。

```

159     elif prev_intersection:
160         bisectors[i] = self.get_bisector(i, in_point=prev_intersection.position)
161         intersections.append(prev_intersection.position)
162         perpendiculars.append(edges[(i+1)%len(edges)].get_perpendicular(next_intersection))
163     elif next_intersection:
164         bisectors[i] = self.get_bisector(i, in_point=next_intersection.position)
165         intersections.append(next_intersection.position)
166         perpendiculars.append(edges[i].get_perpendicular(prev_intersection))

```

図 55 片方にのみ交点が存在する場合

両方とも存在しないケースは、図形が限られており、難易度が格段に跳ね上がるため、本プログラムでは触れないことにした。

このプログラムでは、三角形のように、二等分線が一点で交わるような図形（中心を持つ図形）では全ての辺に二等分線が降ろせるようになってしまっているので、三角形のみはプログラムに成功したため、以下図 56 は三角形のみを場合分け(if 文)し、先のプログラムに else 文を使う事で分離した。三角形のコード内容は、考察にて論じたことと変わらないため、割愛する。

```
144     if len(bisectors) == 3:
145         bisectors[i] = self.get_bisector(i, in_point=next_intersection.position)
146         intersections.append(next_intersection.position)
147         perpendiculars.append(edges[0].get_perpendicular(next_intersection))
148
149     else:
150         if prev_intersection and next_intersection:
151             if prev_distance >= next_distance:
152                 bisectors[i] = self.get_bisector(i, in_point=next_intersection.position)
153                 intersections.append(next_intersection.position)
154                 perpendiculars.append(edges[(i+1)%len(edges)].get_perpendicular(next_intersection))
155             else:
156                 bisectors[i] = self.get_bisector(i, in_point=prev_intersection.position)
157                 intersections.append(prev_intersection.position)
158                 perpendiculars.append(edges[i].get_perpendicular(prev_intersection))
159         elif prev_intersection:
160             bisectors[i] = self.get_bisector(i, in_point=prev_intersection.position)
161             intersections.append(prev_intersection.position)
162             perpendiculars.append(edges[(i+1)%len(edges)].get_perpendicular(next_intersection))
163         elif next_intersection:
164             bisectors[i] = self.get_bisector(i, in_point=next_intersection.position)
165             intersections.append(next_intersection.position)
166             perpendiculars.append(edges[i].get_perpendicular(next_intersection))
167         else:
168             bisectors[i] = self.get_bisector(i, length=0)
169
170     return bisectors, intersections, perpendiculars
```

図 56 三角形の場合分け

この関数の return に bisectors, intersections, perpendiculars を返せば、関数 `get_bisectors()` が完成する。

```

124     def get_bisectors(self):
125         edges = list(map(Segment, self.get_segments()))
126         perpendiculars = []
127         bisectors = []
128         intersections = []
129         angles = []
130
131         for vertex_index in range(len(self.vertices)):
132             bisectors.append(self.get_bisector(vertex_index))
133
134         for vertex_index in range(len(self.vertices)):
135             angles.append(self.get_angle(vertex_index))
136
137         for i, target_bisector in enumerate(bisectors):
138             prev_bisector = bisectors[i-1]
139             next_bisector = bisectors[(i+1)%len(bisectors)]
140
141             prev_intersection, prev_distance = target_bisector.get_intersection(prev_bisector)
142             next_intersection, next_distance = target_bisector.get_intersection(next_bisector)
143
144             if len(bisectors) == 3:
145                 bisectors[i] = self.get_bisector(i, in_point=next_intersection.position)
146                 intersections.append(next_intersection.position)
147                 perpendiculars.append(edges[0].get_perpendicular(next_intersection))
148
149             else:
150                 if prev_intersection and next_intersection:
151                     if prev_distance >= next_distance:
152                         bisectors[i] = self.get_bisector(i, in_point=next_intersection.position)
153                         intersections.append(next_intersection.position)
154                         perpendiculars.append(edges[(i+1)%len(edges)].get_perpendicular(next_intersection))
155                     else:
156                         bisectors[i] = self.get_bisector(i, in_point=prev_intersection.position)
157                         intersections.append(prev_intersection.position)
158                         perpendiculars.append(edges[i].get_perpendicular(prev_intersection))
159
160             elif prev_intersection:
161                 bisectors[i] = self.get_bisector(i, in_point=prev_intersection.position)
162                 intersections.append(prev_intersection.position)
163                 perpendiculars.append(edges[(i+1)%len(edges)].get_perpendicular(next_intersection))
164             elif next_intersection:
165                 bisectors[i] = self.get_bisector(i, in_point=next_intersection.position)
166                 intersections.append(next_intersection.position)
167                 perpendiculars.append(edges[i].get_perpendicular(prev_intersection))
168             else:
169                 bisectors[i] = self.get_bisector(i, length=0)
170
171         return bisectors, intersections, perpendiculars

```

図 57 class Polygon

工場①：svg_utils.py : SVG ファイルの操作を専門とする工場

i . 任意の多角形の作図

以前、長方形の一般化を行ったが、それ以外の図形の作図も行いたいということで、*create_svg()*という任意の多角形が作図できる関数を作る。今回のファイルではインポートしなければいけないライブラリやファイルが多くある（図 58 参照）

```

1  import svgwrite
2  from lxml import etree
3
4  from objects import Polygon

```

図 58 svg_utils.py でインポートするライブラリ&ファイル

引き数として頂点座標とファイル名を受け取っている関数 *create_svg()*をつくる。最初は、前にもやったキャンバスの指定である（7行目）。8行目では、*frame_points = [(50,50),(550,50),(550,550),(50,550)]*であると示している。次にキャンバスに座標が *frame_points* であり、色は黒である多角形（一辺 500 の正方形）を追加する（9行目）。これは折り紙のフレームだと考える。10行目以降は *for* 文で *points* と言うリスト（頂点の座標群）から *vertices*（変数）を取り出している。そして、キャンバスに、座標が引き数として渡した頂点座標を、色は緑色である多角形を追加している（切り取りたい図形の色は緑色と決めた）。これにより頂点の入力により任意の図形の描画が行えるようになった（※ 図形の描画はリスト：*points* の0番目から順に線が引かれていく）

```

6 def create_svg(points,svg_path):
7     canvas=svgwrite.Drawing(svg_path, profile='tiny')
8     frame_points = [(50,50), (550,50), (550,550), (50,550)]
9     canvas.add(canvas.polygon(points=frame_points, stroke=svgwrite.rgb(0,0,0), fill='none'))
10    for verticies in points:
11        canvas.add(canvas.polygon(points=verticies, stroke=svgwrite.rgb(0,255,0), fill='none'))
12    canvas.save()

```

図 59 折り紙のフレームと多角形を描画する関数

ii. ファイルの読み取り

この *svg_utils.py* という工場で最終的に作りたいものはキャンバスに *get_bisectors()* で描けるようになった二等分線の交点までを全て追加できる関数である。だが、それを実行するには数学の証明でいう定理がまだ足りない。まずは、キャンバス内に含まれるすべての要素を解析、判別しなければならない。だいぶ前に作った *svg_parsing()* では解析した線が折り紙のフレームなのか切り取りたい図形なのかまではわからない。よってグレードアップした解析ツールを作る。いきなり飛躍した関数は、難しいので、一旦扱いにくい文字列から扱いやすい数字に直す関数を作る。

図 60 では、まず引き数として *element*（要素）を受け取っており、この関数はリスト内の要素から *point*（点座標）に変換する（間に、文字列から浮動小数点数に直す手順も含まれる）ためのものである。*points* と言う空箱を作り、次の *for* 文で順番に獲得した頂点座標等を格納する。*for* 文の中身は、*element.get('points')* で *element* から *points*（この *points* は点の座標を表す）というキーに対応する値（文字列）を取得している。*.strip()* は取得した文字列の前後についている空白を除いている。*.split(' ')* は前にも述べたように「（スペース）で文字列を分割している。ここで得られた値は *point* に取得され、*for* 文でループが起きるため、全ての要素を文字列で取得可能である。次の 17 行目は括弧の中からバラしていく。まず取得した *point* をカンマで分割する（座標はカンマによって分けられているため、この操作に

より一つ一つの点座標がわかる）。次に map(float….)で前のステップで得られた文字列を浮動小数点数に変換している（文字列から数字に変換）。そして、得られた値をリストにして、points に append している。

```
14  def get_points_from_element(element):
15      points = []
16      for point in element.get('points').strip().split(' '):
17          points.append(list(map(float, point.split(','))))
18  return points
```

図 60 element (文字列) から point (浮動小数点数) への変換を行う関数

次に本命の *read_svg()*を作る。引き数には svg_path を、なぜなら SVG ファイルのみから全情報を取得してほしいからである。関数 etree.parse()で指定されたファイルを解析し、ElementTree オブジェクトを生成している。ElementTree オブジェクトでないと xml 形式（マークアップ言語の一つで tag で囲まれた要素を定義する）のファイルにアクセスできないため関数 etree.parse()が必要となる。

green, black, blue, red の空箱を作る。tree.iter()は xml データを反復処理するための関数であり、for 文で xml データを処理させるために使用した。color は element から “stroke”に対応する値を取得している（図 40、16 行目と原理は同じ）。ここから色で場合分けをする。

If 文で color が緑の時は、element を *get_points_from_element()*で浮動小数点数に変換し（頂点座標の獲得）、この座標を使って新しい Polygon オブジェクトを作成している。そして、得られた多角形（オブジェクト）をリスト : green に *append* している。同様の事を green の他に black, blue, red でも行う。この 4 色は、緑：切り取りたい図形、黒：折り紙のフレーム、青：谷折り線、赤：山折り線 を表している。この情報を文字の列として、return する。

```
return {
    "cut": green,
    "frame": black,
    "mountain_fold": red,
    "valley_fold": blue
}
```

図 61 return の中身

```

20  def read_svg(svg_path):
21      tree = etree.parse(svg_path)
22      green, black, blue, red = [], [], [], []
23      for element in tree.iter():
24          color = element.get("stroke")
25          if color=='rgb(0,255,0)':
26              green.append(Polygon(get_points_from_element(element)))
27
28          elif color=='rgb(0,0,0)':
29              black.append(Polygon(get_points_from_element(element)))
30
31          elif color=='rgb(255,0,0)':
32              red.append(Polygon(get_points_from_element(element)))
33
34          elif color=='rgb(0,0,255)':
35              blue.append(Polygon(get_points_from_element(element)))
36      return {
37          "cut": green,
38          "frame": black,
39          "mountain_fold": red,
40          "valley_fold": blue
41      }

```

図 62 ファイルをより詳しく解析する関数

iii. 情報の添付

`svg_path` が完璧に解析可能な関数が作成できたので、次はキャンバスに書き込むプログラムを作成する。キャンバスに図形を追加するというのは、SVG情報を変更するということである。だが、`read_svg()`の時と同様にいきなり一つの関数にまとめるのは、混乱を招く上、元のファイルに変更を加えるので、最小限のステップで済ませたい。よって、まずは与えられた SVG 情報を使用して新しい SVG ファイルを作り、そこに変更を加える関数を作る。

まずは引き数として、`svg_info` と `svg_path` を受け取っている。`svg_path` はキャンバスの指定に必要であり、`svg_info` は情報そのものである。

`frame_obj` は `svg_info` の `frame` キーに対応しており、且つゼロ番目の要素であるため、`[0]`も追加で付け加えている。`svg_info` から抽出した情報の座標(`frame_obj.vertices`)を使って黒色の多角形をキャンバスに追加している。

他の `cut_obj_list`, `mf_obj_list`(`mf` は mountain fold の略), `vf_obj_list`(`vf` は valley fold の略)は一つの要素だけでは表せない(`frame_obj` は一つの正方形(多角形)として捉えられるため一つの要素だけで表現可能)ため、“`_list`”が後ろについている。他の `frame_obj` との相違点は `for` 文が使用されると言うことだろう。複数の要素が含まれるリストから一つずつ要素を取り出すには `for` 文を使わなければな

らない。後は、獲得した单一の要素の座標を使用して、それぞれの色の多角形をキャンバスに追加している。これにより別のキャンバスで異なる色の輪郭線を持つ多角形が作図できるようになった。

```
43 def create_svg_from_info(svg_info, svg_path):
44     canvas = svgwrite.Drawing(svg_path, profile='tiny')
45
46     frame_obj = svg_info["frame"][0]
47     canvas.add(canvas.polygon(points=frame_obj.vertices, stroke=svgwrite.rgb(0,0,0), fill="none"))
48
49     cut_obj_list = svg_info["cut"]
50     for cut_obj in cut_obj_list:
51         canvas.add(canvas.polygon(points=cut_obj.vertices, stroke=svgwrite.rgb(0,255,0), fill="none"))
52
53     mf_obj_list = svg_info["mountain_fold"]
54     for mf_obj in mf_obj_list:
55         canvas.add(canvas.polygon(points=mf_obj.vertices, stroke=svgwrite.rgb(255,0,0), fill="none"))
56
57     vf_obj_list = svg_info["valley_fold"]
58     for vf_obj in vf_obj_list:
59         canvas.add(canvas.polygon(points=vf_obj.vertices, stroke=svgwrite.rgb(0,0,255), fill="none"))
60
61     canvas.save()
```

図 63 新たなキャンバスで多角形を描画する関数

この *create_svg_from_info()* という関数を使用して、元のファイルを編集する関数を作成する。引き数として *svg_path* と折り紙のフレーム以外の線のデフォルト値を *False* (無いと仮定) として受け取っている。元々 *False* にしておけば、ある時は *if* 文で存在する時という場合わけを、無いときはそのままにしておけば、関数上無いものとしてカウントされるので、非常に便利なフレーズである。*svg_info = read_svg(svg_path)* は *read_svg()* により *svg_info* を獲得しており、これにより関数 *create_svg_from_info()* が使用可能になる (引き数として *svg_info* を受け取っているため)。次の 3 つの *if* 文は、引き数として受け取ったフラグ (*cut*, *mountain_fold*, *valley_fold*) がある (*False* でない) ならば、それに対応する多角形を SVG 情報 (“*cut*”, “*mountain_fold*”, “*valley_fold*”) に追加している (SVG 情報の更新)。(フラグとはブール型のデータ(*True* または *False*)を持つ変数や値) (データには色んなタイプがあり、ブール型はその一つ、他には数値データ、文字列データ、リスト、オブジェクトがある)

更新した SVG 情報を使用して、*create_svg_from_info()* で新しい SVG ファイルを作成している。作った *svg_path* はこの関数では使わないが、目標の関数を作る時に使う。よって、この関数の return は *svg_info* だけとした。

```

62  def add_svg(svg_path, cut=False, mountain_fold=False, valley_fold=False):
63      svg_info = read_svg(svg_path)
64      if cut:
65          svg_info["cut"].append(cut)
66      if mountain_fold:
67          svg_info["mountain_fold"].append(mountain_fold)
68      if valley_fold:
69          svg_info["valley_fold"].append(valley_fold)
70      create_svg_from_info(svg_info, svg_path)
71      return svg_info

```

図 64 指定されたキャンバス（オリジナル）に多角形を描画する関数

iiiまでで作った関数を使用して、objects.py の内容を描画する関数を作る。引き数は *add_svg()*と同様に *svg_path* だけで、*svg_info* は *read_svg* を使って定義する。得られた *svg_info* をプリントする（あってもなくても良い、だがこのプログラムによって正しい情報が獲得できているかを確認できる）。この関数は指定された図形に折り線を入れるためのものであるから、欲しい事前情報は描画した多角形のみである。よって、for 文で SVG 情報（“cut”のみ）から obj(オブジェクト)を順に取り出し、bisectors, intersections, perpendicular は obj を *get_bisectors()*に代入して得られた値であることを示している（この関数の return は bisectors、intersections と perpendicular）。

bisectors というリスト内には *bisector_obj* が沢山入っている。これを for 文で取り出し *add_svg()*で *svg_path* はこの関数でも *svg_path*、でも *mountain_fold* には *bisector_obj* を代入している。この操作により、キャンバスには *bisector_obj* が *mountain_fold*（赤色の線、山折り線）として描画される。現状求められている、各頂点の為す角の二等分線は全て、山折り線として描けるようになった。

次も *bisectors* の時と同様に *perpendicular_obj* を取り出している。相違点は *valley_fold*（谷折り線）として描画されることであろう。それ以外は同様のコードである。

最後に *intersections*(二等分線の交点)を結べば直線骨格の完成である。現段階では交点が 3 つ以上ある時の“その後”はプログラム化できていないので、一旦交点群をまた別の多角形として捉えることで、交点同士を順番に結ぶコードにしている。

```

73 def add_bisectors(svg_path):
74     svg_info = read_svg(svg_path)
75     print(svg_info)
76     for obj in svg_info["cut"]:
77         bisectors, intersections, perpendiculars = obj.get_bisectors()
78         for bisector_obj in bisectors:
79             add_svg(
80                 svg_path=svg_path,
81                 mountain_fold=bisector_obj,
82             )
83         for perpendicular_obj in perpendiculars:
84             add_svg(
85                 svg_path=svg_path,
86                 valley_fold=perpendicular_obj,
87             )
88         add_svg(
89             svg_path=svg_path,
90             mountain_fold=Polygon(intersections)
91         )

```

図 65 折り図を作成する関数

工場②で作成した二つのメインの関数 (*create_svg()*= 与えられた情報からキャンバスを創造する関数、 *add_bisectors()*= 図形に折り線の情報を追加する関数) をまとめた関数 (*one_complete_straight_cut()*) により、この関数ひとつで折り線の追加が可能になった。

```

93 def one_complete_straight_cut(vertices, svg_path="output.svg"):
94     create_svg(vertices, svg_path)
95     add_bisectors(svg_path)

```

図 66 頂点情報から折り図を作成する関数

※GUI のコードの説明は、数学的関連性が薄い為ここでは省略します。

参考文献 Bibliography

『【NumPy 入門】ベクトルの大きさ（ノルム）を計算する np.linalg.norm』 ,
<https://www.sejuku.net/blog/73783>, 参照日 2023/08/13

『xml の基本的な使用方法』, <https://python.keicode.com/advanced/xml-lxml-1.php>, 参照日 2023/08/17

『行列式 $|A|=ad-bc$ の幾何学的意味』, https://note.com/dr_kano/n/n94dfeb646cad, 参照日 2023/08/20

『【ゲーム数学】2つのベクトルのなす角度を求める』, <https://nekojara.city/math-2vector-angle>, 参照日 2023/09/07

『Python で 三 角 関 数 を 計 算 (sin, cos, tan, arcsin, arccos, arctan) 』 ,
<https://note.nkmk.me/python-math-sin-cos-tan/>, 参照日 2023/09/07

『Origami Simulator』, <http://apps.amandaghassaei.com/OrigamiSimulator/>, 参照日 2023/08/22

Erik D. Demaine and Joseph O'Rourke : Geometric Folding Algorithms, Linkages, Origami, Polyhedra. Cambridge University Press, July, 2007

エリック・D・ドメイン、ジョセフ・オルーク：幾何的な折りアルゴリズム-リンクエージ、折り紙、多面体 近代科学社、2009 年

朝日新聞 GLOBE 数学という力 2010/02/01 号

岩井宏朋：一刀切り - One Complete Straight Cut - の理論と実践 普通部労作展 2021

岩井宏朋：一刀切り - One Complete Straight Cut - の理論と実践 2 普通部労作展 2022