平成 13 年度　卒業研究

地理情報システムのための JAVA ライブラリ JTS の実装と評価

2002 年　2 月　12 日

島根大学 総合理工学部 数理・情報システム学科

計算機科学講座（田中研究室）

s98490－B　　森岡　慶彦

目次

第１章　はじめに

現在、情報分野において発展著しい方面の１つに、Geographic Information System(GIS)、地理情報システムがある。GIS は、国の形状、山の標高、河川の経路、植物分布、人口密度、降水量などの地理データ、あるいは都市、幹線、送電線、ガス管のような人工物データなどを扱うことができる。そして、蓄えたデータから、ある領域に関する情報を手に入れること、情報同士の比較を行うことができる。

ほとんどの地理情報は、地球上の点と領域に関するものであり、幾何学的問題が起こりうる。また、システムの元となるデータ量はほとんどの場合膨大な量であり、そこから必要となるデータを探すことは大変な作業である。しかも、必要なデータを実用的時間内で手に入れなければ、システムとして機能しない。

これらの理由より、GIS には優れた計算幾何学的アルゴリズムを実装し、システムの効率化を試みる必要がある。

一方、計算幾何学が世間の注目を集めるようになったのは、1978 年にカーネギーメロン大学の I.Shamos 氏の論文「Computational Geometry」が公表された頃から、とされている。そのときから今日まで、非常に多くの技術やアルゴリズムが研究・開発され、多くの成果を残している。

そのため現在のシステム設計者は、自らの用途に合わせた最適の技術またはアルゴリズムを、蓄積された多くの計算幾何学的研究成果の中から選ぶことができる。

さて、最近数あるプログラム言語の中で評価の高まっているものが Java 言語である。Java 言語は、言語自体のわかりやすさ、豊富な機能、Web ブラウザとの連携、プラットホームの柔軟性などから、プログラミング分野のみならず、通信、教育、娯楽など様々な分野で利用されている。

そのような近年の状況の中で、Java Topology Suite (JTS)の開発が公表された。これは Java 言語で構成された地理情報システム向けの Application Programming Interface(API)であり、高性能・高品質・信頼性の高い機能、アルゴリズムを多々採用している。現在は開発途上であるが、将来的には広く製品に搭載され、普及していくことが期待される。

本研究では、地理情報システム、計算幾何学、Java 言語の 3 要素の集大成である JTS をテーマとする。研究目的は、JTS を地理情報システムに組み込む場合、どの程度の性能が見込めるか、を調査することにある。

具体的には、

1)　JTS を実装した Java プログラムを作成し、テストを行う

2)　1)のテスト結果を参考に、JTS の機能・アルゴリズムを解析する

といった方法で JTS にアプローチし、総合的な性能を評価する。

○本論に入る前に

Geometry という単語の訳は幾何学である。一方 OpenGIS Consortium Inc.では、幾何学的な地形を表す用語として Geometry を選んだ。GIS 分野では Geometry を図形と訳している。

本論では、幾何学的話題では Geometry を幾何学、GIS に関する話題では Geometry を図形、という意味でとらえている。

(なお OpenGIS とは、Open Geodata Interoperability Spaecification の略であり、空間データ相互運用性仕様という意味である。つまりこれは、GIS の相互運用性と基幹業務システムへの統合を目的とした、Open GIS Consortium Inc.(OGC ：非営利団体)が定める GIS ソフトウェアの標準仕様のことである。)

また本研究では、研究に際し調べたアルゴリズム、または研究データを表現するために、以下のような図を頻繁に使用する。

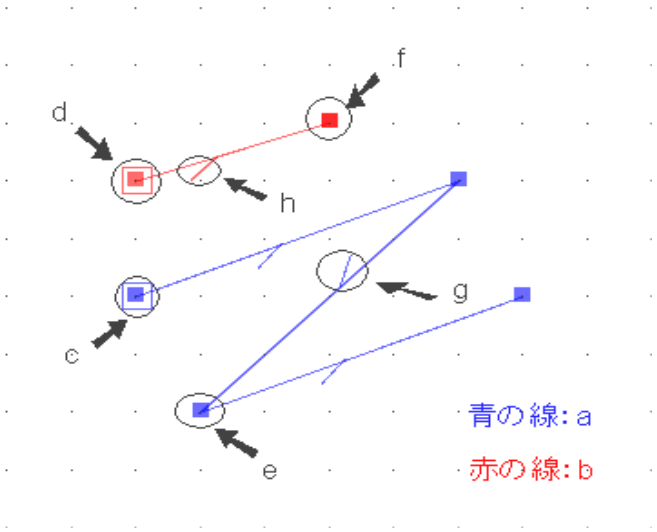


図 1.1. 本研究の図の説明

表 1.1. 図 1.1の記号の意味

記号	意味	記号	意味
a	図形A	e	図形Aの頂点
b	図形B	f	図形Bの頂点
c	図形Aの始点	g	図形Aの方向ベクトル
d	図形Bの始点	h	図形Bの方向ベクトル

この図は、Vivid Solutions 社が開発した JTS TestBuilder というツールで作成した図である(JTS TestBuilder の詳細は 4.2 節を参照)。

図形の始点(図形の座標値の一番先頭の値)は、頂点記号のさらに外側を正方形で囲んでいる。方向ベクトルとは、現在の頂点から次の頂点への方向を表している。

本論では、図の見やすさを優先するため、頂点や方向ベクトルを表示しない図も使用している場合があることを最初に断っておく。

第２章　JTS

2.1.　JTS とは

JTS とは **J**ava **T**opology **S**uite の略であり、カナダの Vivid Solutions 社によって開発されている Java API の事である。JTS は、OpenGIS Consortium Simple Fertures Specification for SQL(SFS)の定義に従って作成されており、基本的 2D 空間アルゴリズムを実装し、GIS 関連のソフトウェアに組み込まれることを目的として開発されている。

JTS は、より効率の良い処理と安定性を追求し、現在も改良が続けられているところである。

また JTS はオープンソースであり、自由に改良し、用途に合わせた利用を行うことが許されている。

2.2.　JTS の特徴について

JTS は以下の特徴を持っている。

- OpenGIS Consortium Simple Fertures Specification for SQL(SFS)で定義された空間的モデル、メソッドを使用している
- 2D 空間アルゴリズム Binary Predicates、Spatial Analysis Methods を実装している
- どのような命令をあらゆる図形に対して実行しても、形状的・図形的に正確な結果を返すような堅固なモデルを採用している
- JTS は実用に耐えるように、また開発者が改良しやすいように、高性能アルゴリズムと鮮明なコードで実装されている
- GIS 分野で一般的な形式である、Well-Known Text(WKT)フォーマットをサポートしている

2.2.1. Spatial Data Model

Spatial Data Model とは、定義されている図形のモデルのことである。

JTS では、以下のモデルが定義されている。

表2.2.1.1. JTSでサポートされるSpatial Data Model

図形名	内容
Point	点(1図形のみ)
MultiPoint	点(2図形以上)
LineString	折れ線(１図形のみ)
LinearRing	折れ線(閉じたLineString)
MultiLineString	折れ線(2図形以上)
Polygon	多角形(1図形のみ)
MultiPolygon	多角形(2図形以上)
GeometryCollection	上記図形の混合

次に、JTS で定義されている全 Spatial Data Model を示す。

図 2.2.1.1. Point

図 2.2.1.2. MultiPoint

図 2.2.1.3. LineString

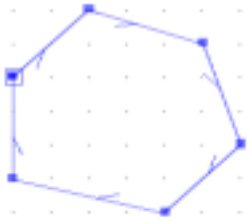


図 2.2.1.4. LinearRing

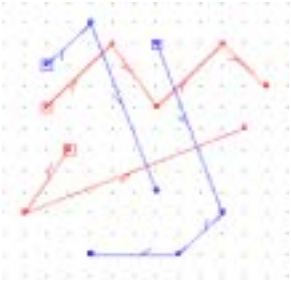


図 2.2.1.5. MultiLineString

図 2.2.1.6. Polygon

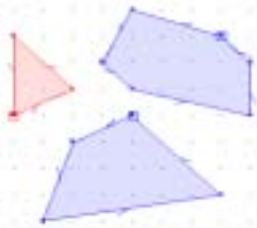


図 2.2.1.7. MultiPolygon

図 2.2.1.8. GeometryCollection

2.2.2. Binary Predicates

Binary Predicates は、ある図形に対して問いかけを行い、その問いかけに合う部分が存在するかどうかを調べ、存在するならば True、存在しなければ False で結果を返す関数である。

JTS では、以下の述部が定義されている。

表2.2.2.1. JTSでサポートされるBinary Predicates

述部名	機能
Equals	2つの図形が同じであるか
Disjoint	2つの図形が離れているか
Intersects	2つの図形が交差しているか
Touches	2つの図形が接しているか
Crosses	一方の図形が他方を横切っているか
Within	一方の図形が他方の内部にあるか
Contains	一方の図形が他方を内部に含んでいるか
Overlaps	2つの図形が重なっている部分があるか

2.2.3. Spatial Analysis Methods

Spatial Analysis Methods は、Spatial Operation を実行するメソッドである。Spatial Operation とは、ある図形に対して問いかけを行い、その問いかけ条件に一致する部分が存在するかどうかを調べ、結果を Geometry 型(Spatial Data Model 型)で抽出する操作のことである。

JTS では、以下のメソッドが定義されている。

表2. 2. 3. 1. JTSでサポートされるSpatial Analysis Methods

メソッド名	機能
Intersection	2つの図形の共通集合を図形として返す
Union	2つの図形の和集合を図形として返す
Difference	一方の図形の他方の図形でない部分を返す
Symmetric Difference	2 つの図形の和集合から共通集合を除いた部分を返す
Convex Hull	図形の凸包を返す
Buffer	図形の指定した範囲内の領域を返す

以下に、各メソッドの結果を図で示す。(結果は黄色領域)

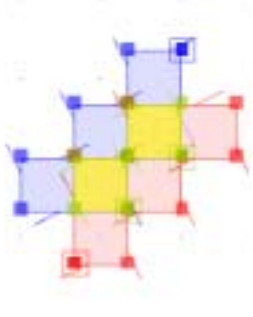


図 2.2.3.1. Intersection

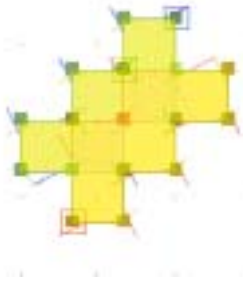


図 2.2.3.2. Union

図 2.2.3.3. Difference(B でない A)

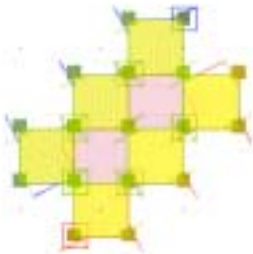


図 2.2.3.4. Symmetric Difference

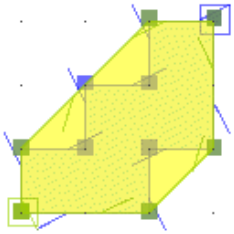


図 2.2.3.5. Convex Hull(A の Convex Hull)

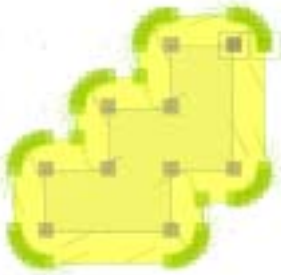


図 2.2.3.6. Buffer

2.2.4. Well-Known Text

JTS では、Well-Known Text(WKT)フォーマット形式でのファイル入出力が認められている。WKT は Open GIS Consortium (OGC)によって定義されている、地理情報システムや SQL 等で図形をテキストで表現する方法である。これをサポートするソフトウェア同士ならば、お互いで情報のやりとりをすることができる。

次に、WKT の一例を示す。

●座標値((100,100),(150,100),(150,150),(100,150))である LineString

・・・LINESTRING(100 100, 150 100, 150 150, 100 150)

●座標値((100,100),(150,100),(150,150),(100,150))である Polygon :

・・・POLYGON((100 100, 150 100, 150 150, 100 150)

第３章　JTS が使用する技術・アルゴリズム

3.1. 計算幾何学的アルゴリズムと実装の問題

計算幾何学の歴史は古く、同じ問題を解く場合でも、多くの技術・アルゴリズムが存在する。GIS 等において計算幾何学アルゴリズムを用いる場合、どの技術・アルゴリズムを用いるかにより、性能に大きな差がでる。やっかいなことに、計算幾何学的には優れた技術・アルゴリズムであっても、プログラム化が困難なもの、資源を大量に消費してしまうもの、実行時間が長すぎるもの、コーディングに向かないなどの理由から、工学的・実用的視点から見ると、最適な技術・アルゴリズムでないものもある。

システムに最適な技術・アルゴリズムを探し、実装することは、システム設計において重要な問題である。

ここでは、JTS が採用している様々な技術・アルゴリズムについて説明する。

3.2. Monotone chains

Monotone chains は、図形の交差する部分を判定する際に使用されているアルゴリズムである。Monotone chains を用いることで、図形の交差部分がどこにあるかを検索する作業を高速に実行できるようになる。

まず、chain についての定義を示す。

定義：chain $C=(u_1,u_2,\cdots,u_n)$ とは、点集合$\{u_1,u_2,\cdots,u_n\}$、
枝集合$\{(u_i,u_{i+1}) \mid i=1,\cdots,n-1\}$のグラフである。

これを元に、Monotone chain は以下のように定義される。

定義：直線 L に垂直であるどの直線も、チェーン $C=(u_1,u_2,\cdots,u_n)$と
ちょうど 1 点で交わるとき、Cは Lに対して Monotone（単調）であるという。

つまり Monotone chains とは、u_1から u_nまで同じ方向を向いている折れ線であるといえる。

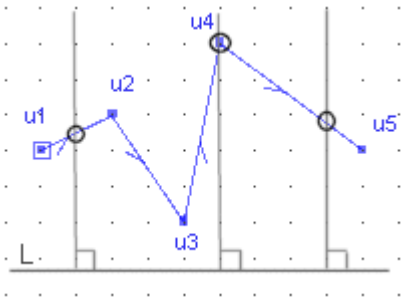


図 3.2.1. Monotone chain

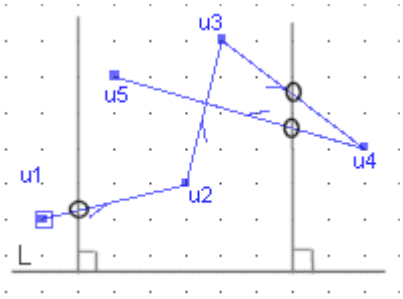


図 3.2.2. Monotone chain でない chain

JTS では、Monotone chains の定義も含めて、新たに **Monotone chains 領域**という概念を設定し、利用している。

以下が、Monotone chains 領域の定義である。

1. Monotone chains 領域内ではその図形自身は決して交差しない
2. Monotone chains 領域のどのような隣接する部分集合のエンベロープ(最小および最大の (X,Y) 座標によって形成されている境界を作る図形)であろうとも、いくつかの部分集合の終点のエンベロープである

1 から、Monotone chain 領域内で図形自身が交差するかをテストする必要はなく、探索時間が減ること、2 から、Monotone chains に沿って 2 分探索することを許す、すなわち探索アルゴリズムを使用できること、が保証される。

以下に、Monotone chains 領域作成方法を示す。(n を頂点数とする)

Ⅰ. i=2 とする。$i \geqq n$ ならⅦへ。そうでないならⅡへ。
Ⅱ. 頂点 u_1 の x、y 座標を $X1$、$Y1$、頂点 u_i の x、y 座標を $X2$、$Y2$ とする。
Ⅲ. $X2-X1$、$Y2-Y1$ の結果の符号を α、β とする。
Ⅳ. u_i の x、y 座標を $x1$、$y1$ とし、u_{i+1} の x、y 座標を $x2$、$y2$ とする。
Ⅴ. $X2-X1$、$Y2-Y1$ の結果の符号を γ、δ とする。
Ⅵ. i+1 し、α と γ、β と δ の符号を比べ、どちらも同じならⅣへ。
そうでないか、$i \geqq n$ ならⅥへ。
Ⅶ. u_1 から u_i までの頂点間を結ぶ辺が Monotone chain であり、u_1 と u_i を対角線とする長方形が Monotone chain 領域となる。

上記のアルゴリズムを全ての頂点に対して実行することで、図形全体の Monotone chains 領域が得られる。

図 3.2.3 に、Monotone chaisn 領域による図形分割を示す。

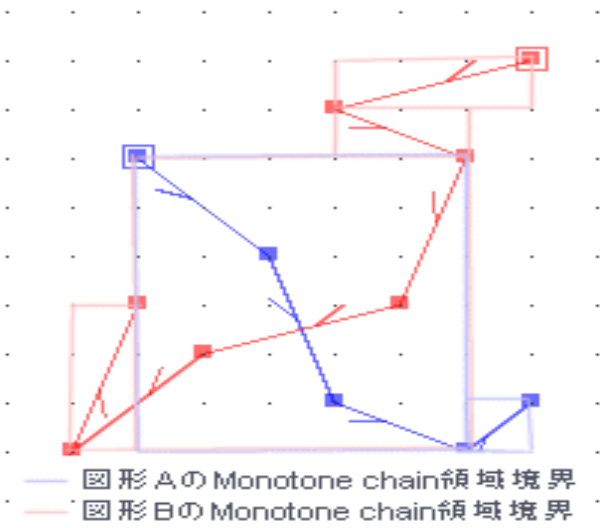


図 3.2.3. Monotone chains 領域による図形分割

Monotone chains は、intersection などの Spatial Operation 等で使用される。

3.3. Intersection Matrix

Intersection Matrix は、図形の特性・図形同士の位置関係を確認する際に使用される。Intersection Matrix は、DE-9IM という手法を核として成り立っている。

まずは DE-9IM の原型である 9IM から説明する。

○ 9IM (9 Intersection Model)

9IM とは、2 つの図形について、図形のどの部分で交差するかを示した数学的な手法のことである。1 つの図形に対し、内部(Interior または I)、境界(Boundary または B)、外部(Exterior または E)の 3 通りの属性がある。以下に、内部、境界、外部の図形毎の範囲を示す。黒い部分が領域である。(Point の Boundary は未定義)

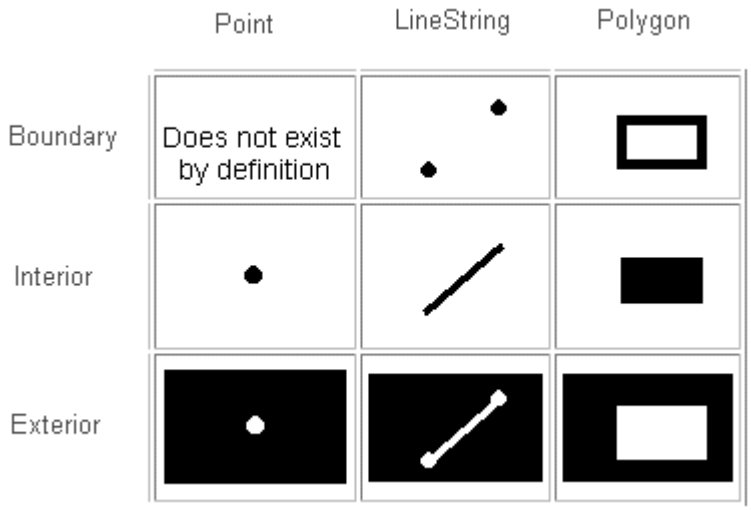


図 3.2.1. 図形毎の内部、境界、外部

2 つの図形の交点モデル(Intersection Model)は、3 通り×3 通りで合計 9 つあることから、9IM と名付けられた。

9IM の欠点は、図形の形状を考慮していないことであり、2 次元以上の図形に対して適用することはできない。

○DE-9IM (Dimensionally Extended 9 Intersection Model)

DE-9IM とは、9IM を次元の点で拡張したものであり、タイプおよび次元の異なる図形同士の位置関係を定義する数学的な手法のことである。

表3. 3. 1. DE-9IMの形式

	Interior	**Boundary**	**Exterior**
Interior	$\dim(I(a) \cap I(b))$	$\dim(I(a) \cap B(b))$	$\dim(I(a) \cap E(b))$
Boundary	$\dim(B(a) \cap I(b))$	$\dim(B(a) \cap B(b))$	$\dim(B(a) \cap E(b))$
Exterior	$\dim(E(a) \cap I(b))$	$\dim(E(a) \cap B(b))$	$\dim(E(a) \cap E(b))$

dim(*x*)は *x* の最大次元を返す関数を意味する。次元の意味は以下の通り。

次元 −1：空（∅）
次元　0：長さも面積もない　・・・　図形 Point、Multipoint に相当
次元　1：長さがある　　・・・　図形 LineString、MultiLineString に相当
次元　2：面積がある　　・・・　図形 Polygon、Multipolygon に相当

なお、次元 0、1、2 の図形をそれぞれ **P**、**L**、**A** で表記する。

DE-9IM の交点モデルは、記号**T**、**F**、**＊**、**0**、**1**、**2** で表現される。

T ≧ dim(*x*) ∈{0, 1, 2}	**0** ≧dim(*x*) = 0
F ≧ dim(*x*) = - 1	**1** ≧dim(*x*) = 1
＊≧ dim(*x*) ∈{-1, 0, 1, 2}	**2** ≧dim(*x*) = 2

DE-9IM は 9 つのパターン値からなる 3×3 行列、もしくは一列のパターン行列で表現される。

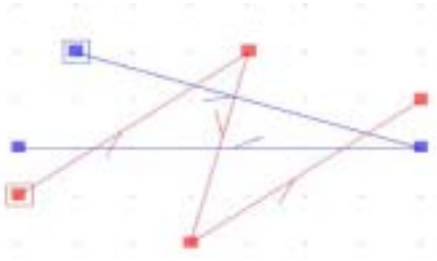


図 3.3.2. 2 つの LineString の例

DE-9IM は $\begin{matrix} 0 \\ F \\ 1 \end{matrix}$ もしくは ***0F1FF0102*** $\begin{matrix} 1 \\ 0 \\ 2 \end{matrix}$

○DE-9IM の用途

JTS では、図形の DE-9IM と JTS が定義した Binary Predicates のパターン(もしくは条件)DE-9IM とをマッチングさせることで、その図形の Binary Predicates を判定させている。マッチング結果が一致(もしくは図形の条件が一致)した場合、Binary Predicates は T(True)を返し、一致しない場合には F(False)を返す。（例： 図 3.2.2 では、Intersects と Crosses が**T**、そのほかは**F** を返す）

表3.3.2. JTSで使用可能な述部とTrueを返す条件

述部名称	引数の次元	Tを返すパターンもしくは条件
Equals	all	T*F**FFF*
Disjoint	all	FF*FF****
Intersects	all	A.intersects(*B*) = ! A.disjoint(*B*)
Touches	P/L, P/A, L/L, L/A, A/A	FT******* or F**T***** or F***T****
	P/P	undefined
Crosses	P/L, P/A, L/A	T*T******
	L/L	0********
	P/P, A/A	undefined
Within	all	T*F**F***
Contains	all	A.contains(*B*) = A.within(*B*)
Overlaps	P/P, A/A	T*T***T**
	L/L	1*T***T**
	P/L, P/A, L/A	undefined

このように IM では、ある図形の DE-9IM と決められたパターンの DE-9IM とをマッチングさせることで、図形がどのような特性を持っているかを判定することができる。JTS ではこの DE-9IM を、Binary predicates をはじめとして、図形を取り扱う場面では必ずと言ってもよいほど使用している。

3.4 .Graham scan

Graham scan は、convexhull(凸包)計算の際に使用されるアルゴリズムである。

○凸包とは

d次元ユークリッド空間 E^d(実数 x_i (i=1,2,・・・,d)の d個組からなり、距離が($\Sigma_{i=1}^{d} x_i^2$)$^{1/2}$で与えられる空間)において、E^d上の領域 Dは、D内の任意の 2 点 q_1、q_2に対し、線分 q_1q_2が D内に完全に含まれるとき、凸であると呼ばれる。

凸包とは、E^d上の点の集合 Sを含む最小の凸領域(の境界)のことである。

図 3.4.1. 凸包作成前

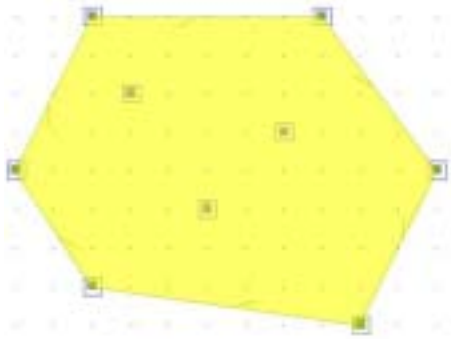


図 3.4.2. 凸包作成後

○凸包アルゴリズム

凸包を求める単純なアルゴリズムは、以下の通り。

1. 端点（凸集合 Sの点であり、S内のどの 2 点 a,bの開線分 ab上にもない点）の集合を求める。
2. これらの点が凸多角形を形成するように並べる。

しかしこのアルゴリズムでは、1 で端点でない不要な点を取り除き、次のソートのための凸多角形の内点を求める計算を行うため $O(N^4)$、2 で内点を軸としてソート（クイックソートの場合）を行うため $O(M\log M)$ となり、全体の計算量は $O(N^4)$ となってしまう。

計算量が $O(N^4)$ のアルゴリズムでは、あまり多くのデータを扱うことはできない。そこで、上記方法とはまったく異なる理論を用いた凸包アルゴリズムが多数考え出された。

Graham scan とは、R.Graham が 1972 年に発表したアルゴリズムであり、凸包の計算に組み込むことで、凸包計算量を $O(M\log M)$ まで減らすことができるのである。

Graham scan を用いた凸包を求めるアルゴリズムは、以下の通りである。

1. 与えられた凸包の内点を求め、それを内点 o とする。
2. 全頂点中、y 座標が最小の点を走査開始点 v_0 とする。
   （y 座標が同じ点がある場合、x 座標が大きな方を選ぶ）
3. o を原点として偏角順（x 軸から反時計回り方向）に点をソートする。
   （ソート後の点を $v_0, v_1 v_2$ ・・・v_n とする）
4. Graham scan を行う　：（図 3.4.3）
   v_{i-1}, v_i, v_{i+1} の 3 点についての内角（内点 o から見た∠v_{i-1}, v_i, v_{i+1}）を調べる（$1 \leqq i < (n-1)$）。
   （ケース 1）内角が 180 度以上（右回り）
   ・・・ v_i を消去し、v_{i-2} , v_{i-1} , v_{i+1} を調べる（逆戻りして調べ直す）。
   （ケース 2）内角が 180 度未満（左回り）
   ・・・ 走査が一歩前進し、次に v_i , v_{i+1} , v_{i+2} を調べる。
   これを全頂点走査（scan）するまで行う。v_0 は消去されず、逆戻りは v_1 まで。
5. 4 終了後、凸包を成す頂点の集合が得られる。

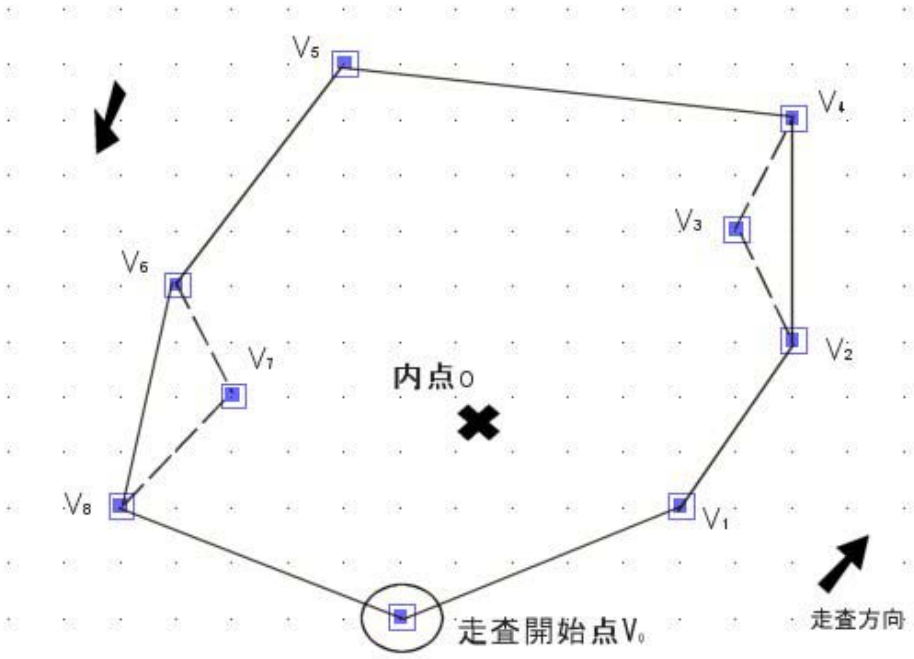


図 3.4.3. Graham scan　（内角∠$v_2 v_3 v_4$ と内角∠$v_6 v_7 v_8$ は 180 度以上のため、v_3 と v_7 を消去）

計算量は、1、2、4 は $O(N)$、3 が $O(N \log N)$ である。よって、全体で $O(N \log N)$ の計算量となる。

JTS では、Spatial Analysis Methods の Convex Hull において凸包を計算するが、その途中で Graham scan を用いている。

第４章　図形作成方法

4.1. 図形作成ツールについて

図形を表現するには座標値を、すなわち x 座標値、y 座標値を与えればよい。単純な図形ならば、人間の手で座標値を考え、作成することができる。しかし、複雑な形の図形を大量に作成・編集・保存するには、何らかの図形作成ツールを使用した方が便利である。

ここでは、本研究において図形を作成する際に使用した、2 つの図形作成ツールについて説明する。

4.2. JTS TestBuilder

JTS TestBuilder は、Vivid Solutions 社が開発した図形作成ツールである。JTS TestBuilder では、GUI を用いた図形作成が可能であり、命令実行結果を画面表示することができる(図 4.2.1 では、2 つの Polugon 図形の intersection を求めている)。また同社は JTS TestRunner というテスト実行ツールも提供しており、JTS TestBuilder で作成した図形から、TestRunner 用のテストファイルを作成する事もできる。

欠点は、一度作成した図形を再編集できない、処理が重い、といった点である。

本研究ではこのツールを、次に紹介するツール makecoordinate の補助的な役割で使用した。

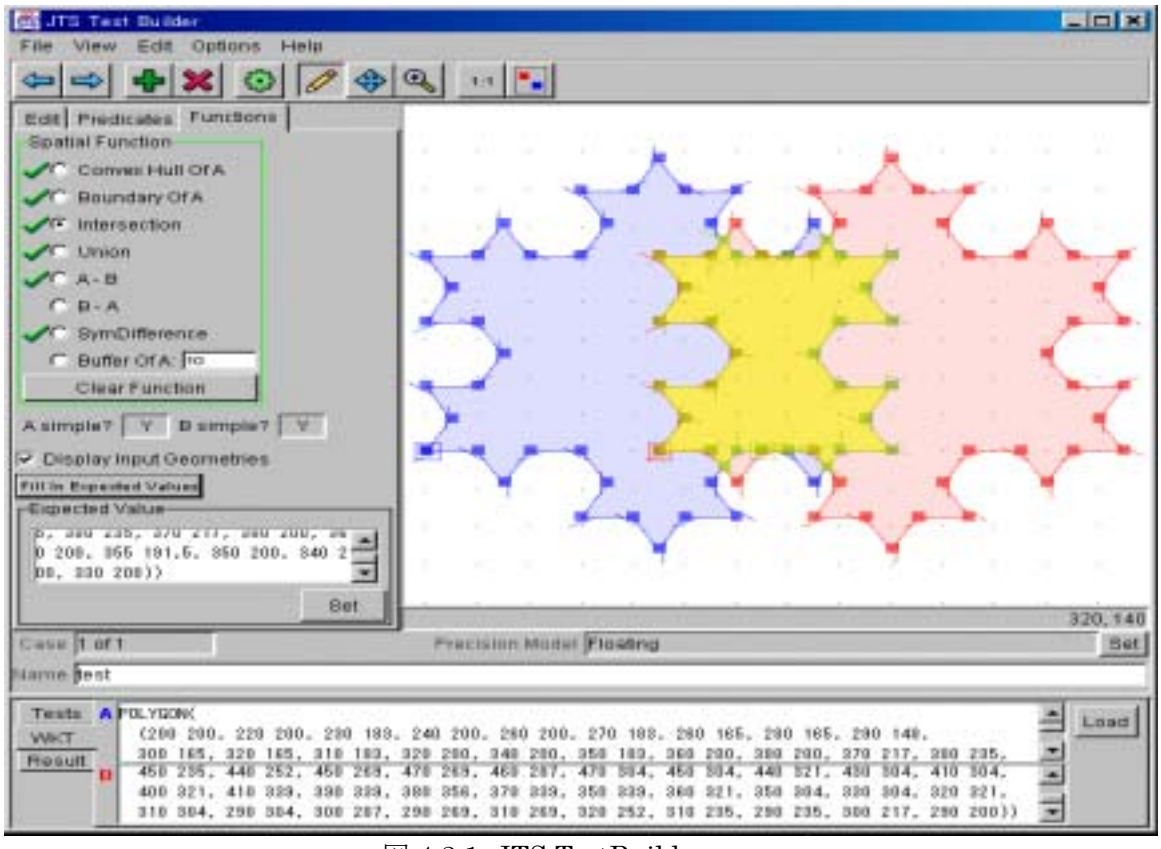


図 4.2.1. JTS TestBuilder

4.3. makecoordinate

図形作成、編集、ファイル入出力、図形画面出力を目的として、本研究のために自作したツールが makecoordinate である。
主な機能は以下の通りである。

●コード全てを Java 言語で作成
●JTS の基本的 2D 空間アルゴリズム(Binary Predicates、Spatial Analysis Methods)を実装
●図形作成アルゴリズムを搭載
・ペントミノ図形(ペントミノパズルの各ピース)を 1～12 個作成するアルゴリズム
・フラクタル図形(コッホ島、クロスステッチ)作成アルゴリズム
・多角形(3～360 角形)作成アルゴリズム
●図形および座標の画面表示可能
●図形編集(頂点の追加・編集・削除、図形の拡大縮小・回転・反転)可能
●ファイル入出力(独自形式、WKT format 形式、JTS TestBuilder 用 XML ファイル形式)可能

欠点は、コマンド方式のプログラムであるため、手軽に図形作成・編集ができない点である。しかし、JTS TestBuilder 用 XML ファイル形式の入出力機能をサポートしているため、このプログラムと JTS TestBuilder の間でファイルのやりとりを行うことで、お互いの欠点を補っている。

また、上記の下線部分のように基本的 2D 空間アルゴリズムの実装を実現している。第 5 章のテストも、このプログラムから実行している。

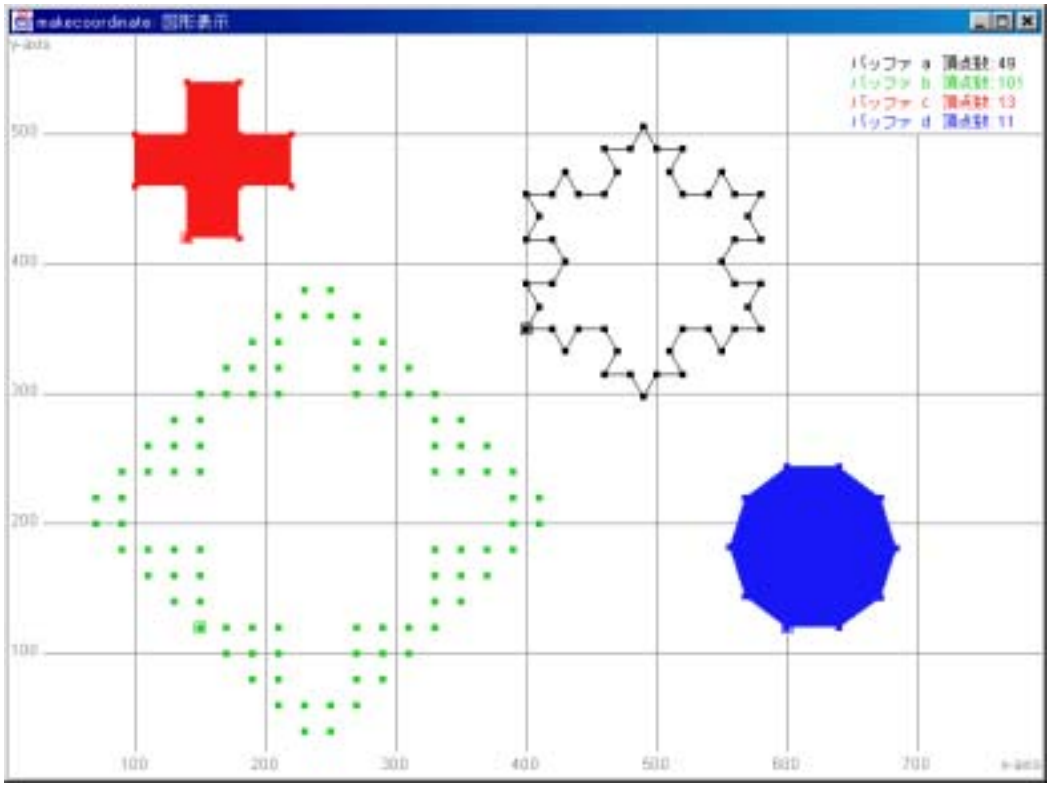


図 4.3.1. makecoordinate　図形表示画面

第５章　テスト

5.1. JTS version 0.9 について

本研究で扱ったのは、JTS version 0.9 である。version 0.9 では、次の注意すべき点がある。

●一部のドキュメントが未完成
●overlay、buffer 関数の結果が不完全である場合がある
●ソースプログラムの整理が不完全

JTS の開発が遅れたため、本研究では JTS の完成版を使用することができなかった。しかし version 0.9 でも、プログラム全体の実行にはそれほど影響がない。
テストでは、JTS version 0.9 で問題なく使用できる命令のみを扱っている。

5.2. フラクタル図形について

本研究のテストデータには、フラクタルという図形を使用した。まずはフラクタル図形について簡単に説明しておく。

フラクタル図形とは、自己相似性を持つ図形である。自己相似性とは、図形の一部分が全体もしくはそれより大きな部分と同じような形になっている、という性質のことである。フラクタル図形の一例として、コッホ島やクロスステッチ、といった図形がある。

○コッホ曲線

次に、コッホ曲線のアルゴリズムを示す。（図 5.2.1 を参照）

ⅰ．開始点 A から長さ r の直線を引く。引き終えたところを点 B とする。
ⅱ．点 B から 60 度の方向に長さ r の直線を引く。引き終えたところを点 C とする。
ⅲ．点 C から-120 度の方向に長さ r の直線を引く。引き終えたところを点 D とする。
ⅳ．点 D から 60 度の方向に長さ r の直線を引く。引き終えたところを点 E とする。

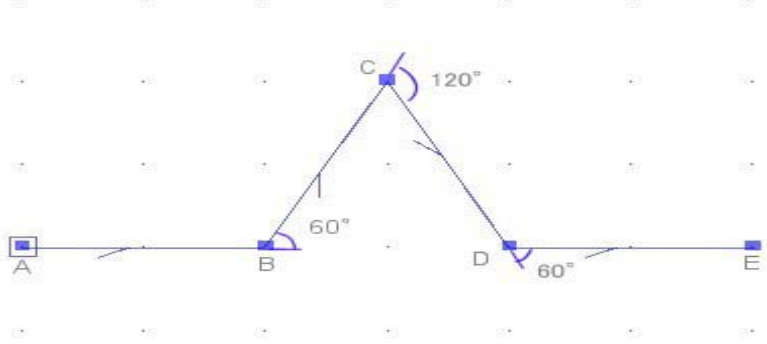


図 5.2.1. 1 次コッホ曲線と作成方法

次数の異なるコッホ曲線を作成する方法は、以下の通りである。

0 次コッホ曲線は、長さ R の直線である。
1 次コッホ曲線は、1 辺 r=1/3×R として、前述アルゴリズムを 1 回実行することで得られる。
2 次コッホ曲線は、1 辺 r=1/9×R として、前述アルゴリズムを 4 回実行することで得られる。
n 次コッホ曲線は、1 辺 r=1/(3 の n 乗)×R として、前述アルゴリズムを 1 + (3×(n−1))回
実行することで得られる。

○コッホ島

コッホ島は 3 辺のコッホ曲線からなる図形であり、以下のアルゴリズムで作成される(図 5.2.2 のⅠ～Ⅲを参照)。

Ⅰ．n 次コッホ曲線を描く。
Ⅱ．-120 度の方向を向き、n 次コッホ曲線を描く。
Ⅲ．120 度の方向を向き、n-1 次コッホ曲線を描く。

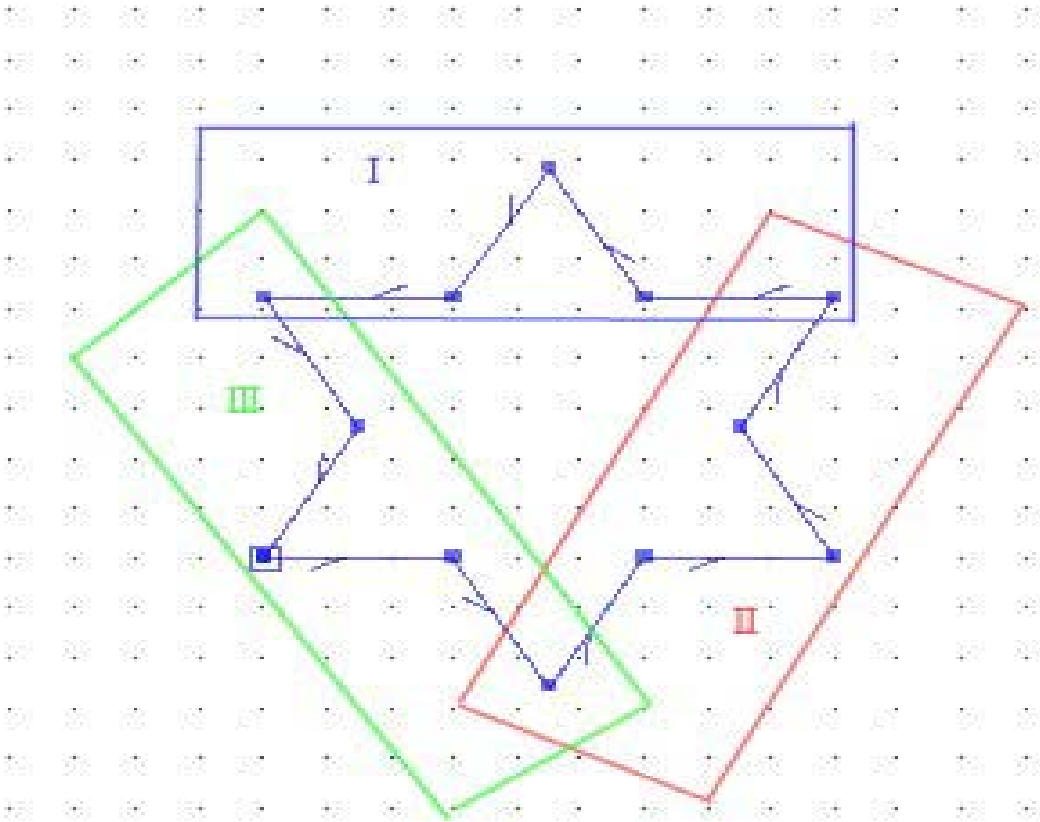


図 5.2.2. 1 次コッホ島と作成方法

○クロス曲線

基本的にはコッホ曲線と同様である。次に、クロス曲線アルゴリズムを示す。(図 5.2.3 を参照)

ⅰ．開始点 A から長さ r の直線を引く。引き終えたところを点 B とする。
ⅱ．点 B から 90 度の方向に長さ r の直線を引く。引き終えたところを点 C とする。
ⅲ．点 C から-90 度の方向に長さ r の直線を引く。引き終えたところを点 D とする。
ⅳ．点 D から-90 度の方向に長さ r の直線を引く。引き終えたところを点 E とする。
ⅴ．点 E から 90 度の方向に長さ r の直線を引く。引き終えたところを点 F とする。

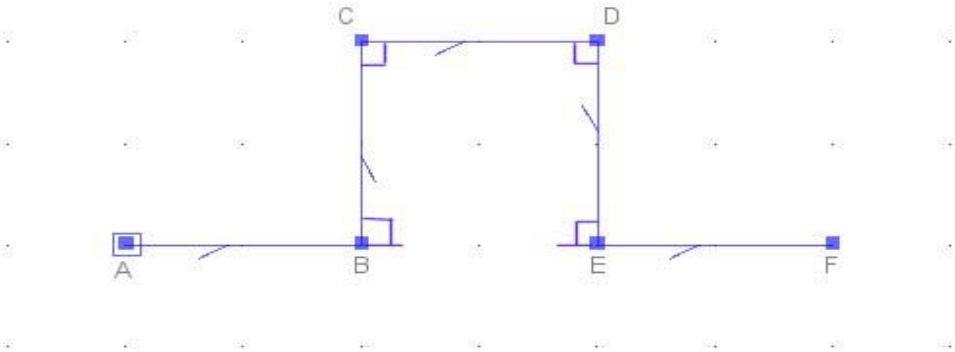


図 5.2.3. 1 次クロス曲線とその作成方法

次数の異なるクロス曲線を作成する方法は、以下の通りである。

0 次クロス曲線は、長さ R の直線である。
1 次クロス曲線は、1 辺 r=1/3×R として、上記アルゴリズムを 1 回実行することで得られる。
2 次クロス曲線は、1 辺 r=1/9×R として、上記アルゴリズムを 4 回実行することで得られる。
n 次クロス曲線は、1 辺 r=1/(3 の n 乗)×R として、上記アルゴリズムを 1+(3×(n−1))回実行することで得られる。

○クロスステッチ

クロスステッチは 4 辺のクロス曲線からなる図形であり、以下の方法で作成される。(図 5.2.4 のⅠ～Ⅳを参照)

Ⅰ．n 次クロス曲線を描く。
Ⅱ．-90 度の方向を向き、n 次クロス曲線を描く。
Ⅲ．180 度の方向を向き、n 次クロス曲線を描く。
Ⅳ．90 度の方向を向き、n 次クロス曲線を描く。

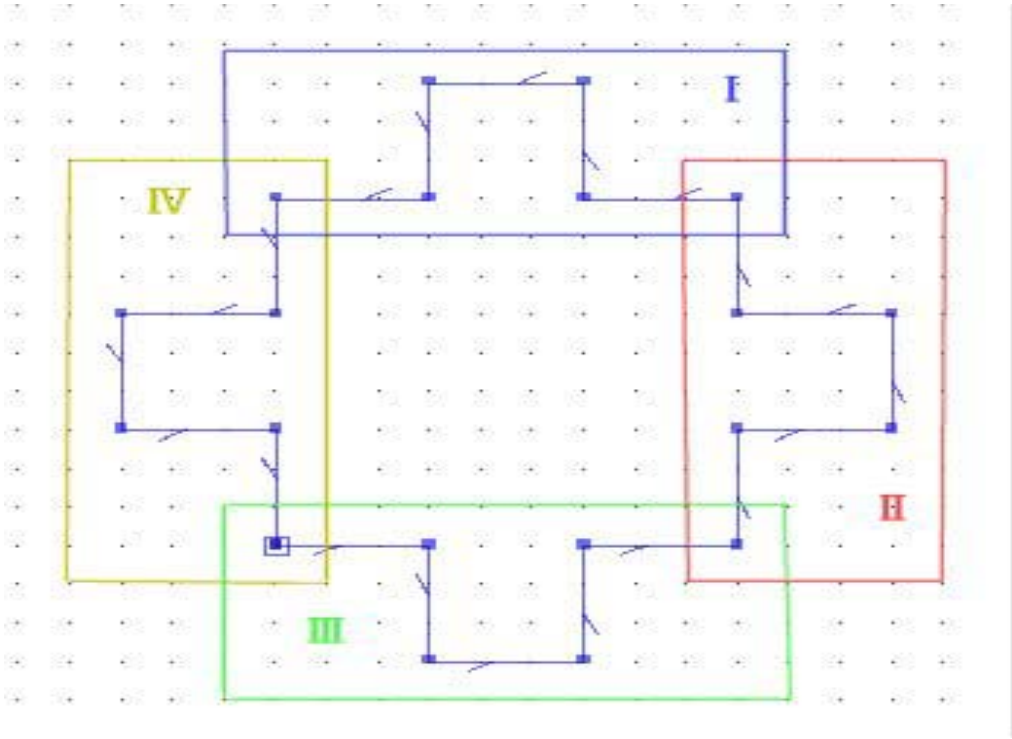


図 5.2.4. 1 次クロスステッチと作成順序

○フラクタル図形を使用する理由

フラクタル図形をテストデータに用いた理由は、頂点数の多い複雑な図形をテストしたかったためである。フラクタル図形は、比較的簡単な再帰的アルゴリズムで多くの頂点数を持つ複雑な図形を作成できるため、本研究のテストに向いていると考えた。

テストデータには、1 次～5 次コッホ島、1 次～4 次クロスステッチを使用する。

5.3. テスト方法・テスト命令

○テスト環境

以下の 2 つのマシンを使用して処理時間を測定した。

表 5. 3. 1. テストマシン

	CPU種類 （周波数）	memoty	OS
マシンA	AMD Duron （950MHz）	256MByte	WindowsXP Professional
マシンB	Intel Pentium Ⅱ(266 MHz)	64MByte	Windows Millennium

マシン 2 台を使用したのは、性能が大きく異なる両マシンにおいて、どのくらい実行時間に差がでるかを参考にしたかったためである。ただしマシン A、B の OS が異なるため、単純な性能比較はしない。

○テスト命令

テストする命令は以下の 3 種類である。

●テスト 1　relate　　(2 つの図形の DE-9IM を求め、文字列として返す)
●テスト 2　intersection　(2 つの図形の共通集合を求め、結果を Geometry 型として返す)
●テスト 3　convexhull　(1 つの図形の凸包を求め、結果を Geometry 型として返す)

テスト 1、テスト 2 は 2 つの図形に対して、テスト 3 は 1 つの図形に対して、命令を実行する。

○テスト方法詳細

テストは、4.3 章において説明した makecoordinate から実行する。両マシンともプログラム実行時には何もオプションをつけない。また、両マシンとも DOS プロンプトから起動した。テスト手順は以下の通り。

①テストデータの格納されたファイルを読み込む
②時間測定を開始
③命令を実行
④時間測定終了
⑤結果を画面出力

なお実行時間は、時間を 10 回測定した平均値をとることにする。使用した Java(SDK1.3)では、Java HotSpot 等の性能向上化技術が採用されているため、次の 2 パターンの実行方法を試した。

・単独実行[Java HotSpot 使用せず] …(プログラム起動、①～⑤実行、終了)
・連続実行[Java HotSpot 使用]　…(プログラム起動、①～⑤実行、続けて①～⑤実行、・・・)

● ○テスト方法詳細 ①について

テストデータは、テスト前にあらかじめツール makecoordinate および JTS TestBuilder にて作成し、ファイルに保存しておく。そのため、テスト時には作成したファイルを読み出すだけでよい。テストデータについては、後述の ○テスト項目内容・テスト図形 を参照。

●テスト実行・時間計測プログラムについて

ここでは、○テスト方法詳細 の②～⑤についてさらに詳しく説明する。

テストする JTS 2D 空間アルゴリズムは、自作プログラム makecoordiante の testJTS クラス において実装されている。

このクラスでは、
・Binary Predicates (contains,crosses,disjoint,equals,intersects,overlaps,touches,within,relate)
・Spatial Operation (convexhull,getboundary,intersection,union,symdifference)
を実行することができる。

次ページより、testJTS クラスにおいて実際にテストを実行しているメソッド judgement のソースプログラム概要を示す(オブジェクト宣言、メッセージ出力、制御文等を一部省略)。

********** judgement メソッド概要 **********

/* このメソッド以前に、1 つ目の図形を data_a に、2 つ目の図形(convexhull, getboundary 以外の Binary predicates または Spatial operation の場合のみ必要)を data_b に格納済み。実行命令も select に格納済み。 */

```
public String judgement(String data_a,String data_b,int select)  {

// 1 つ目の図形 data_a を、WKT 形式の文字列にして Geometry a に格納
   Geometry a = wktRdr.read(data_a);
   Geometry b = a; Geometry c = a;
   String s = new String();
   Date date_start,date_finish;
   double start,finish,time=-1;

// 2 つ目の図形が必要な場合、data_b を WKT 形式の文字列にして Geometry b に格納
   if ((select != 8) && (select != 9)) {
      b = wktRdr.read(data_b); }
// Date オブジェクト作成時点の時刻を得る
      date_start = new Date();
// Date オブジェクトで表される時刻(ミリ秒)を start に格納
      start = date_start.getTime();
                ・(relate 命令の場合の処理)
                  // DE-9IM を求めるメソッド IntersectionMatrix を実行し、結果を im に格納
                      IntersectionMatrix im = new IntersectionMatrix(a.relate(b));
                ・(intersection 命令の場合の処理)
                  // 共通集合計算 intersection を実行し、結果を c に格納
                      c = a.intersection(b);
                ・(convexhull 命令の場合の処理)
                  // 凸包計算 convexHull を実行し、結果を c に格納
                      c = a.convexHull();
// Date オブジェクト作成時点の時刻を得る
      date_finish = new Date();
// Date オブジェクトで表される時刻(ミリ秒)を finish に格納
      finish = date_finish.getTime();
// start –finish を行うことで、命令実行時間 time を得る
      time = finish - start;
//  命令結果を WKT 形式の文字列にして s に格納
        (intersection ,convexhull 命令の場合の処理)   s = wktWriter.write(c);
        (relate 命令の場合の処理)                      s = im.toString();

      System.out.println( /* ここで実行命令名、実行時間 time を画面出力 */ );
      return s;      //実行結果の文字列 s を返す
}
```

****

judgement メソッドにおいて重要な部分は、プログラム太字部分である。これらをまとめると、このメソッドでは主に次のような処理を実行している。

Ⅰ. Date 型オブジェクト **date_start** を作成し、命令実行直前の時刻を得る。
Ⅱ. Date クラスの getTime()メソッドより、オブジェクト **date_start** から時刻を手に入れ、double 型変数 start に格納する。この start が命令実行前の時刻となる。
Ⅲ. 選択した命令を実行する。
Ⅳ. Date 型オブジェクト **date_finish** を作成し、命令実行直後の時刻を得る。
Ⅴ. Date クラスの getTime()メソッドより、オブジェクト **date_finish** から時刻を手に入れ、double 型変数 finish に格納する。この finish が命令実行後の時刻となる。
Ⅵ. finish から start を減算し、結果を double 型変数 time に格納することで、命令実行時間 time を得ることができる。
Ⅶ. time を画面出力する。

以上のことから、テスト命令の実行時間を計測することができる。

5.4.テストデータ

○テスト項目内容・テスト図形

●テスト 1 relate 命令・テスト 2　intersection 命令 テスト内容

relate 命令、intersection 命令ともに同じ図形とテスト項目を使う。こうすることで、同じ 2 つの図形に対する異なる命令において、どのくらい実行時間に差が出るかをチェックできる。
テストは、表 4.2.1.の図形に対して、表 4.2.2.の項目をチェックする。これを(図形番号. テスト項目)の形式で表す(例 3.5 は 3 次コッホ島の図形が完全一致した場合のテスト)。テストデータは全 35 項目。頂点数・図形位置が異なる場合、どのくらい実行時間に差が出るかをチェックする。

表5.4.1. relate・intersection命令テスト図形

番号	図形	頂点数
1	1次コッホ島	13
2	2次コッホ島	49
3	3次コッホ島	193
4	4次コッホ島	769
5	5次コッホ島	3037

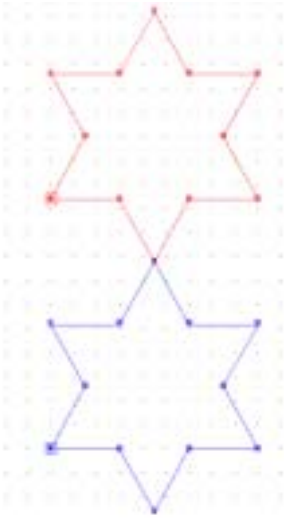
表5.4.2. relate・intersection命令テスト項目

番号	テスト項目
1	一点交差
2	一辺交差
3	お互いの半分で交差
4	一致から β を右に10移動
5	完全一致
6	完全分離
7	一方が一方内に存在

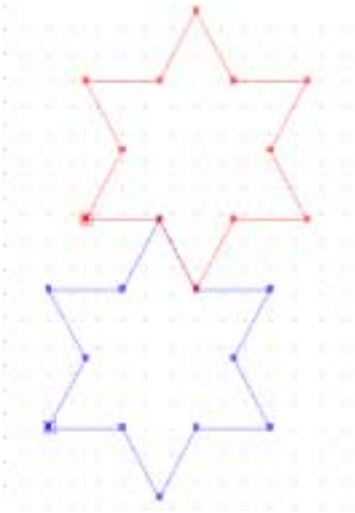
●relate 命令・intersection 命令 テスト項目図形（ベクトル方向は描写せず）

図形属性は全て閉じた LineString とし、次のようなテストデータを用意した。

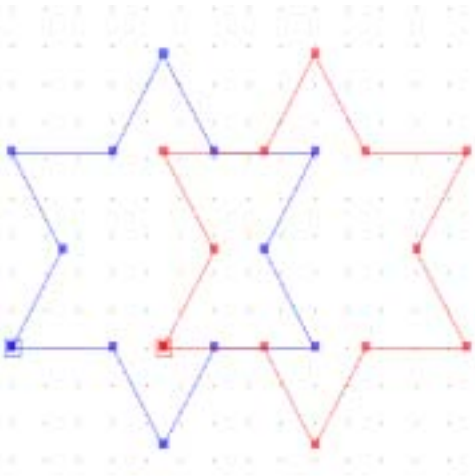
テスト 1.1



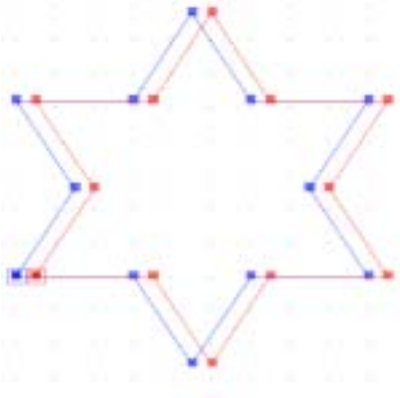
テスト 1.2



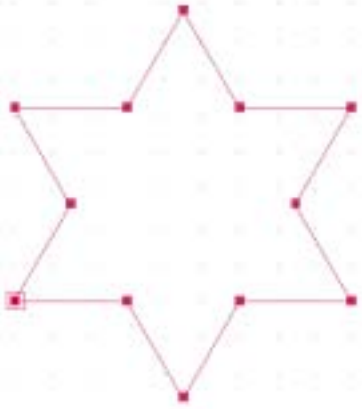
テスト 1.3



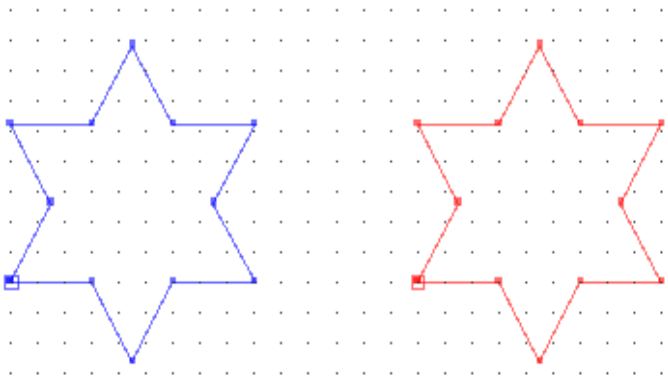
テスト 1.4



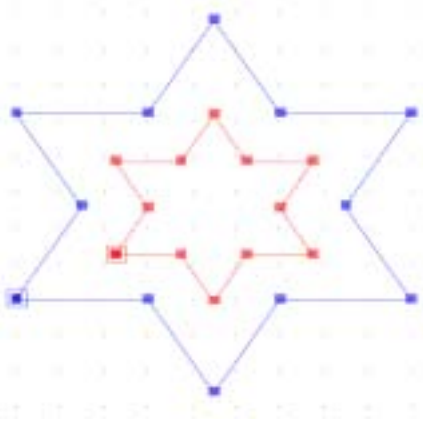
テスト 1.5



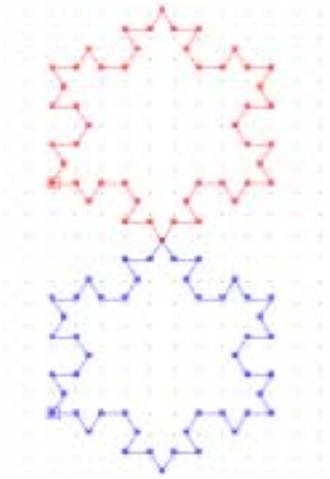
テスト 1.6



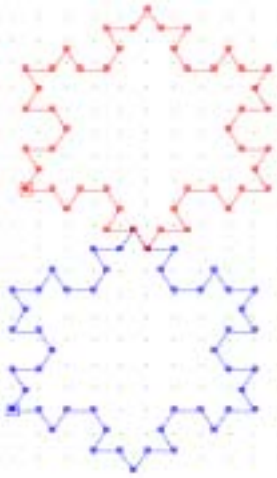
テスト 1.7



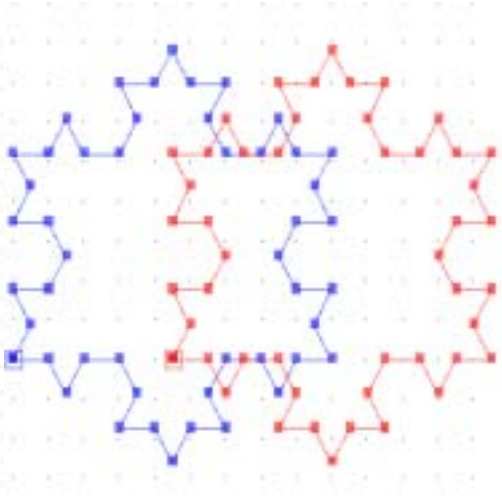
テスト 2.1



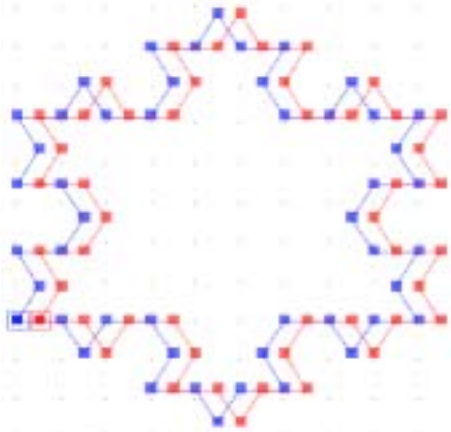
テスト 2.2



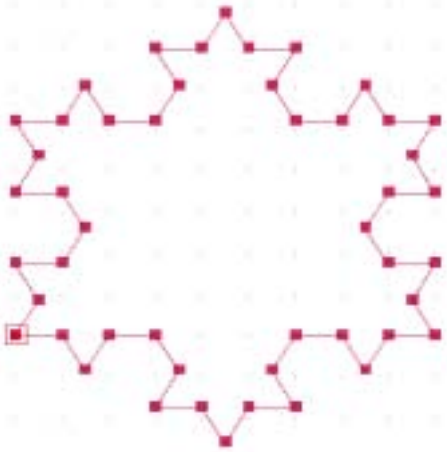
テスト 2.3



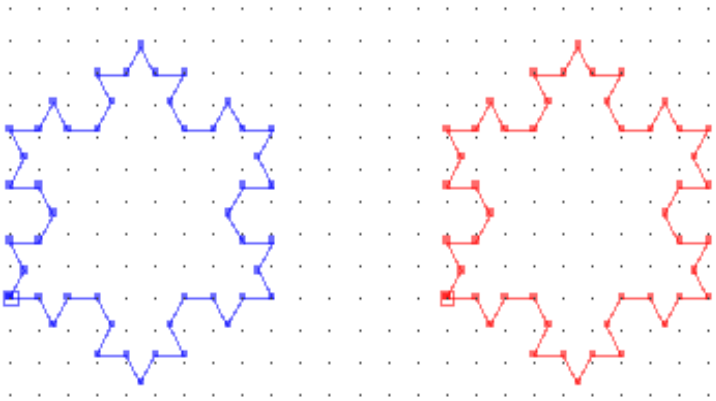
テスト 2.4



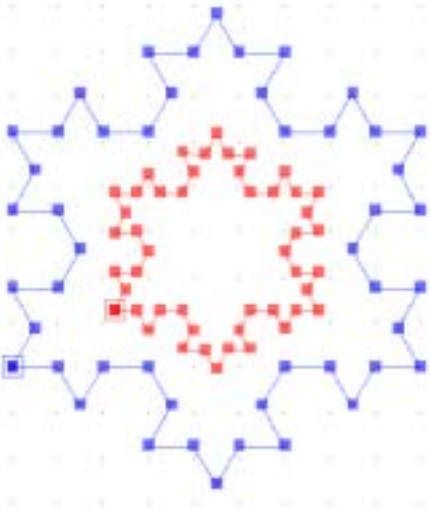
テスト 2.5



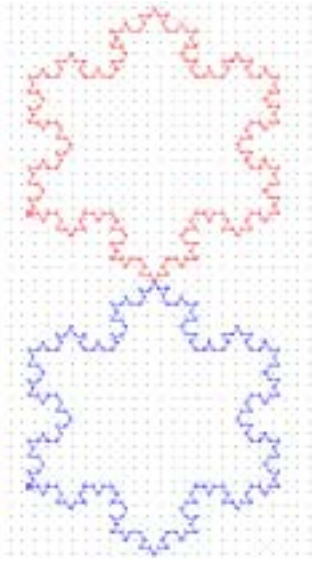
テスト 2.6



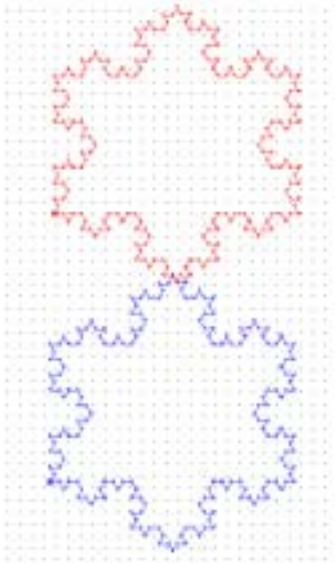
テスト 2.7



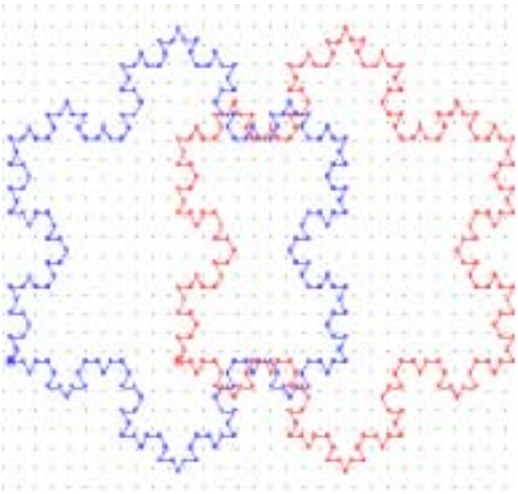
テスト 3.1



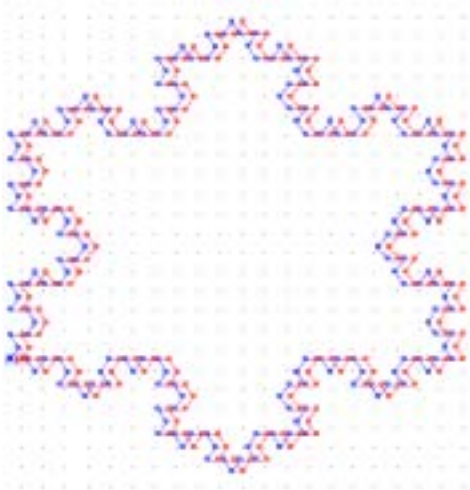
テスト 3.2



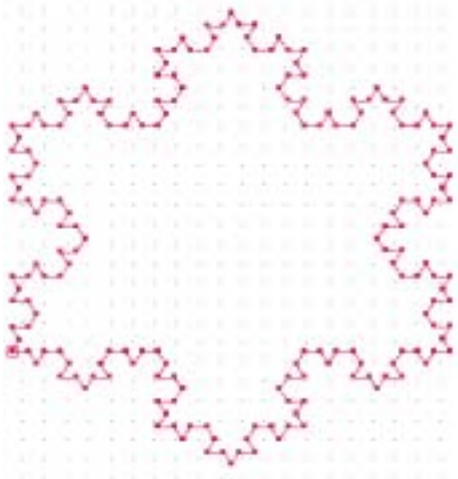
テスト 3.3



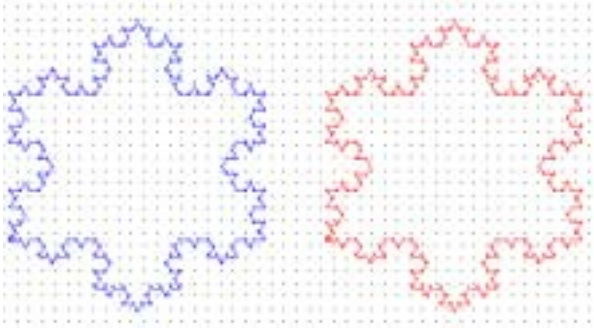
テスト 3.4



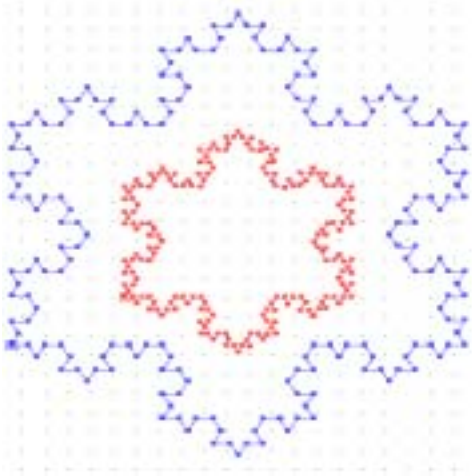
テスト 3.5



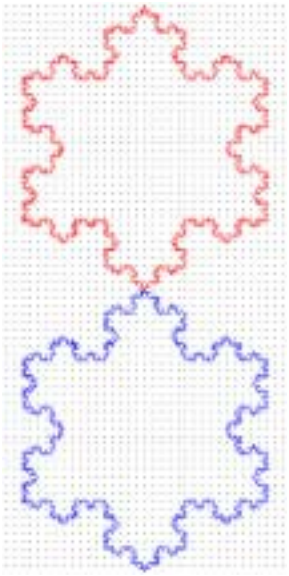
テスト 3.6



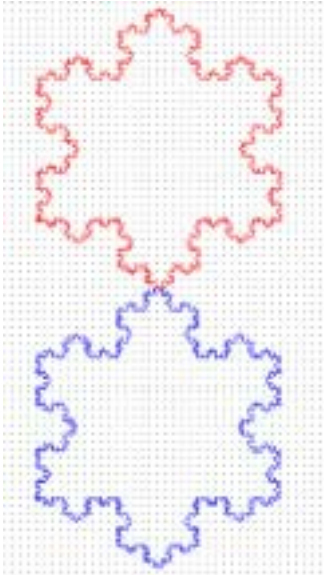
テスト 3.7



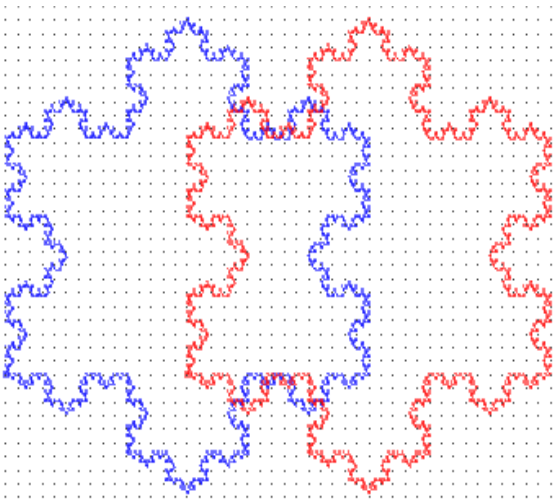
テスト 4.1



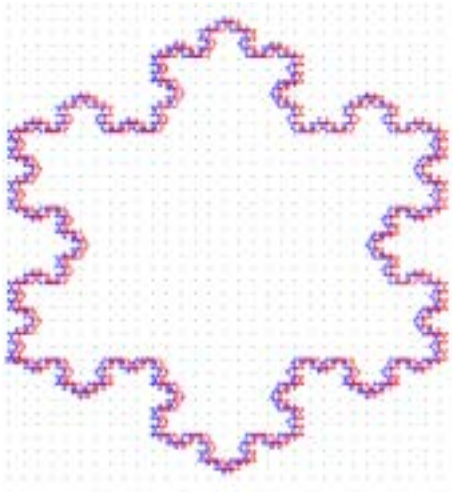
テスト 4.2



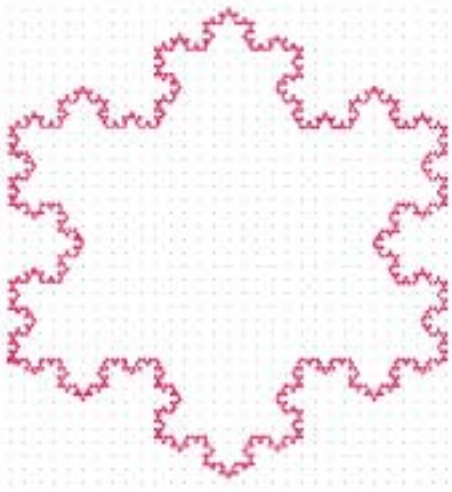
テスト 4.3



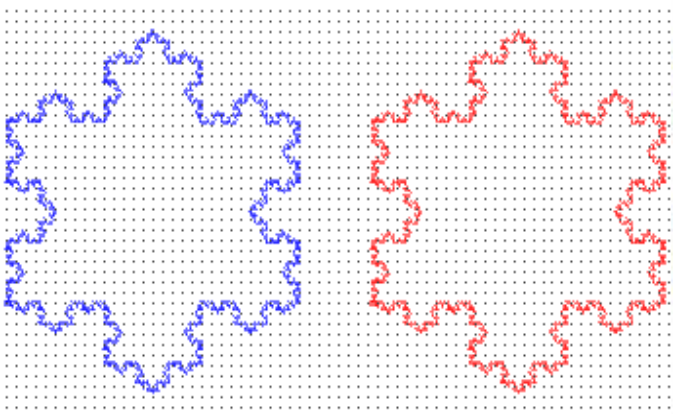
テスト 4.4



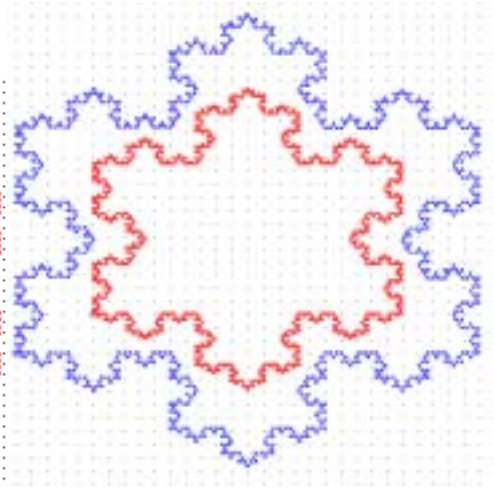
テスト 4.5



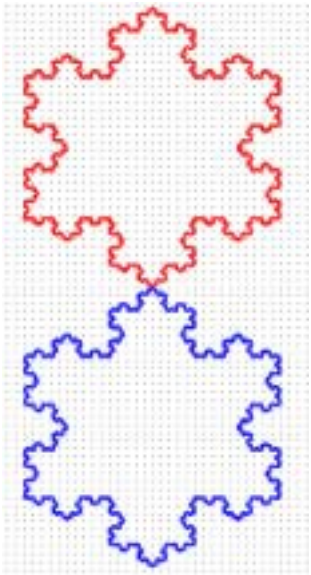
テスト 4.6



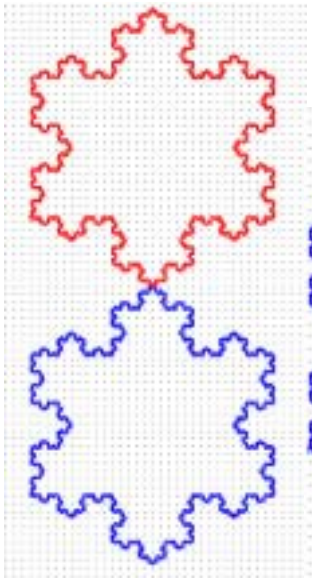
テスト 4.7



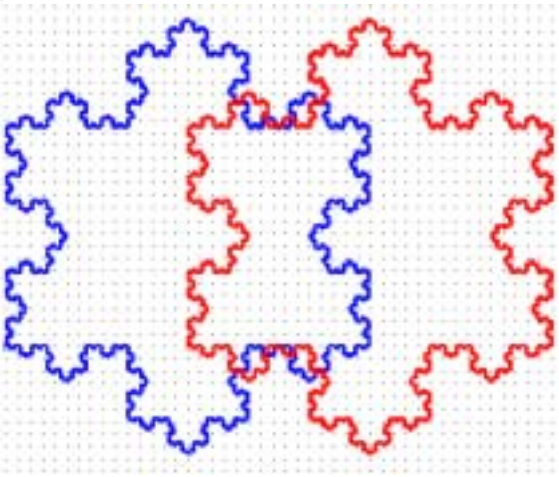
テスト 5.1



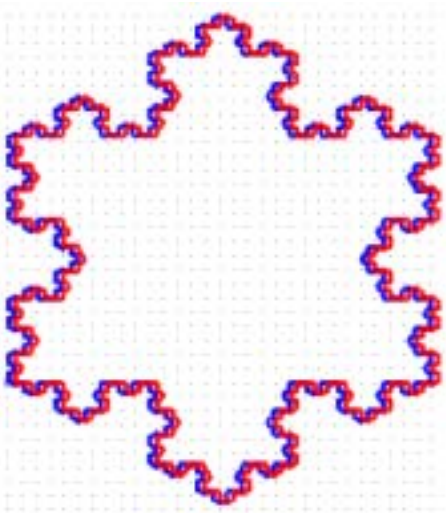
テスト 5.2



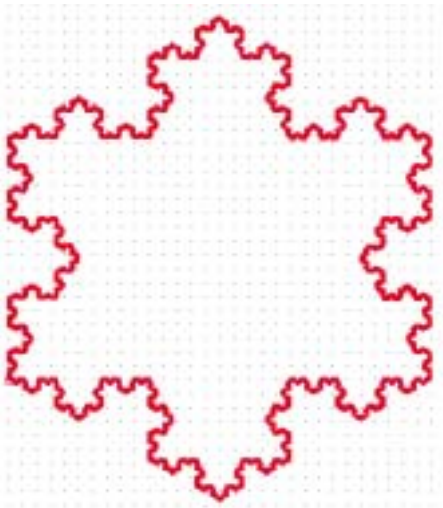
テスト 5.3



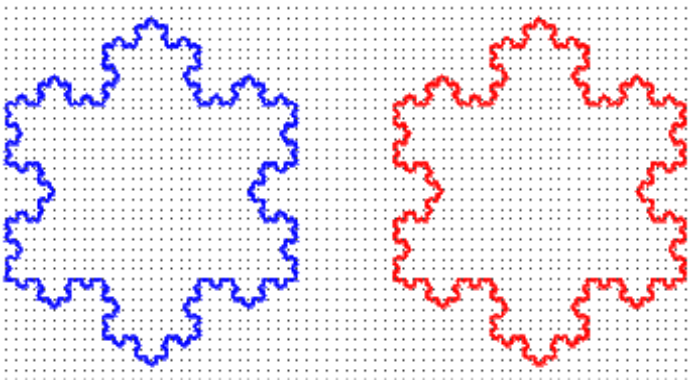
テスト 5.4



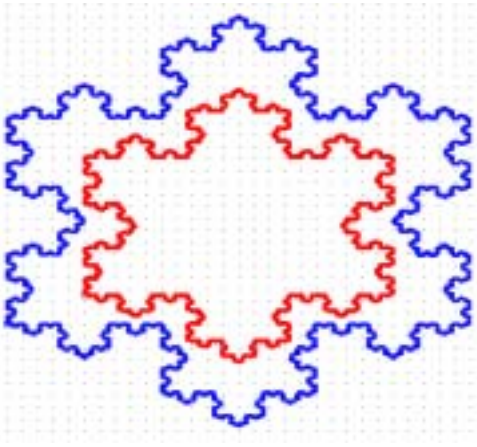
テスト 5.5



テスト 5.6



テスト 5.7



●テスト 3　convexhull 命令 テスト内容

コッホ島を中心にしたテストと、クロスステッチを中心にしたテストの 2 つを用意した。コッホ島・クロスステッチ各次数と同じ頂点数の正方形・大小正方形の図形を作成し、図形の形状によってどのくらい実行時間に差が出るかをチェックする。

テストデータは、コッホ島は全 15 項目、クロスステッチは全 12 項目、合計 27 項目である。

表5.4.3 convexhull命令テスト図形(コッホ島)

番号	図形	頂点数
1	1次コッホ島	13
2	正方形	13
3	大小正方形	13
4	2次コッホ島	49
5	正方形	49
6	大小正方形	49
7	3次コッホ島	193
8	正方形	193
9	大小正方形	193
10	4次コッホ島	769
11	正方形	769
12	大小正方形	769
13	5次コッホ島	3037
14	正方形	3037
15	大小正方形	3037

表5.4.4 convexhull命令テスト図形(クロスステッチ)

番号	図形	頂点数
1	1次クロスステッチ	21
2	正方形	21
3	大小正方形	21
4	2次クロスステッチ	101
5	正方形	101
6	大小正方形	101
7	3次クロスステッチ	501
8	正方形	501
9	大小正方形	501
10	4次クロスステッチ	2501
11	正方形	2501
12	大小正方形	2501

注.　テスト図形の**正方形**とは、図 5.4.1 のような図形である。このような形になったのは、頂点数が多くなると座標値がかなり大きな値となり、図形の作成・編集などが難しくなるので、頂点間の間隔を狭くしたためである。頂点間の間隔が狭いため、隣接した頂点との頂点表示が重なり、太い線の正方形に見える(図 5.4.2 と形は同じ)。

また、テスト図形の**大小正方形**とは、図 5.4.3.のように、内側に小さな正方形が、外側に大きな正方形が描かれた図形である。これは図 5.4.4 のように、一筆書きされている図形である。

図 5.4.1.正方形(頂点数 13)

図 5.4.2. 正方形(図 5.4.1 の頂点間隔を広げたもの)

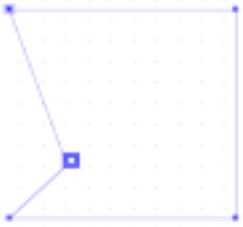


図 5.4.3.大小正方形(頂点数 13)

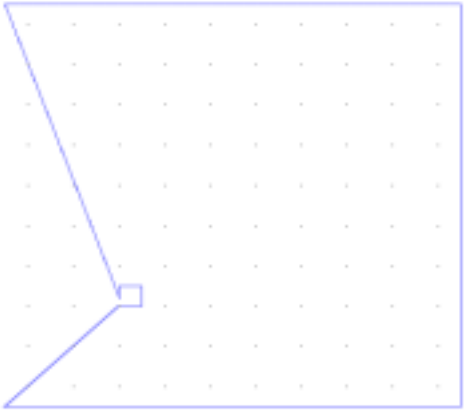
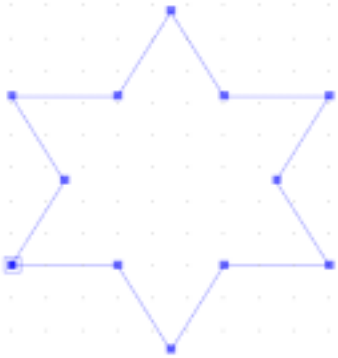


図 5.4.4.大小正方形(図 5.4.3 の頂点を非表示にしたもの)

●convexhull 命令［コッホ島］テスト項目図形（ベクトル方向は描写せず）

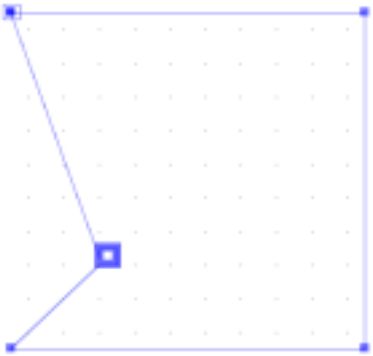
図形属性は全て閉じた LineString とし、次のようなテストデータを用意した。

テスト 1

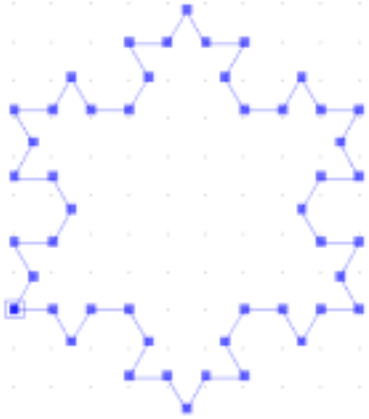


テスト 2

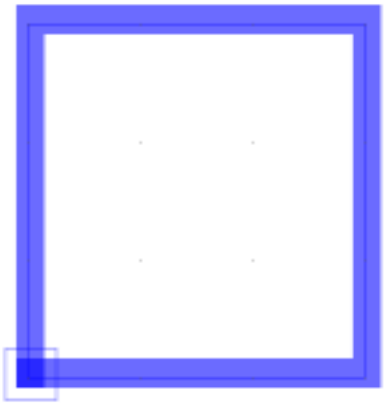
テスト 3



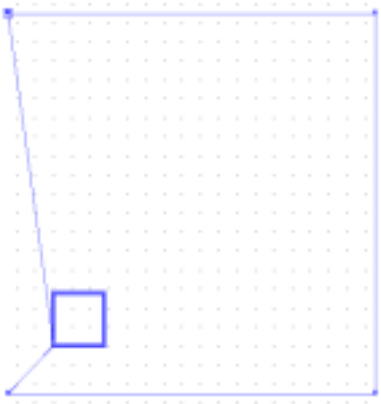
テスト 4



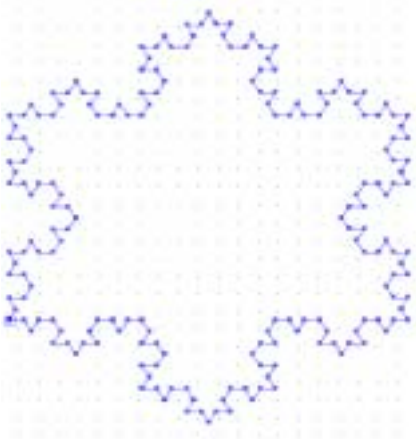
テスト 5



テスト 6

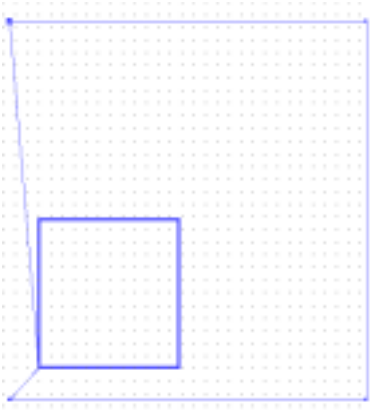


テスト 7

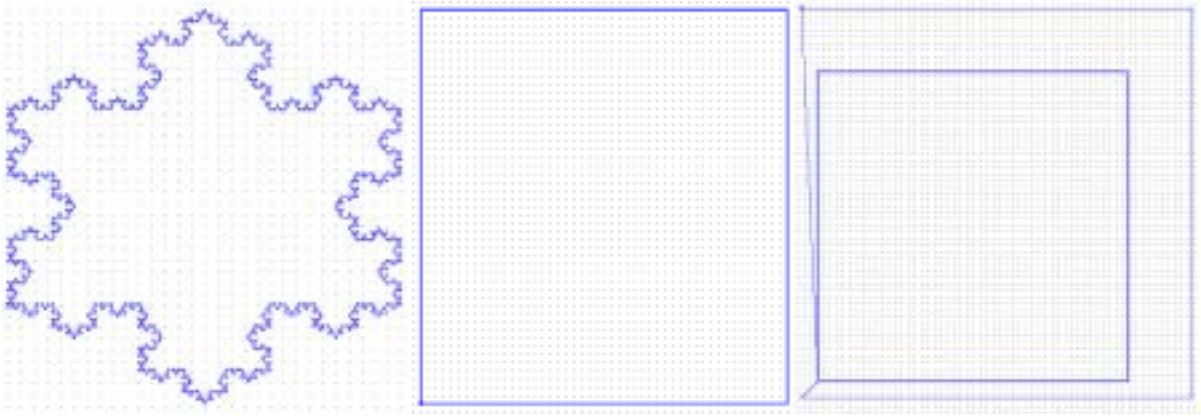


テスト 8

テスト 9



テスト 10　　テスト 11　　テスト 12

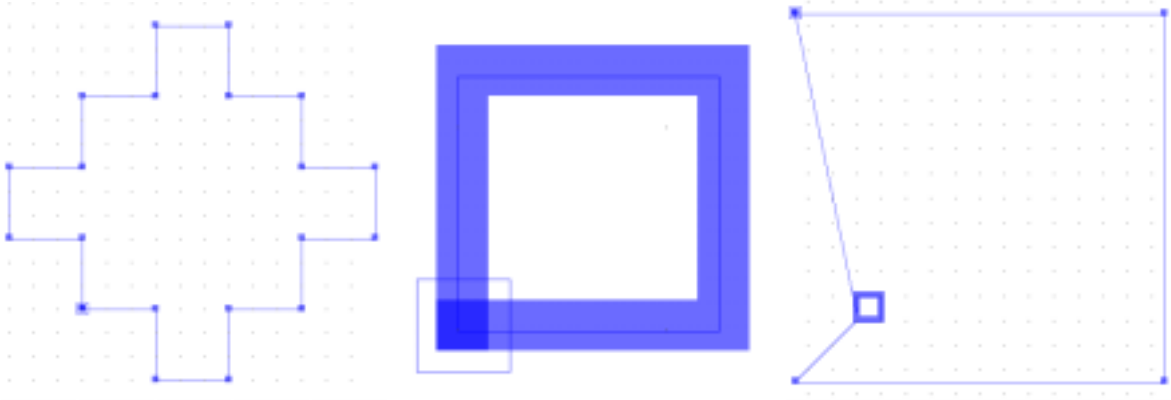


テスト 13　　テスト 14　　テスト 15

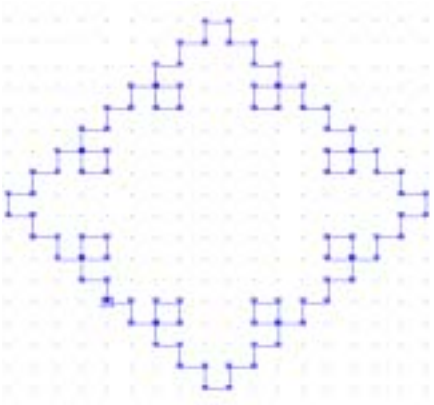
●convexhull 命令［クロスステッチ］テスト項目図形（ベクトル方向は描写せず）

図形属性は全て閉じた LineString とし、次のようなテストデータを用意した。

テスト 1　　テスト 2　　テスト 3

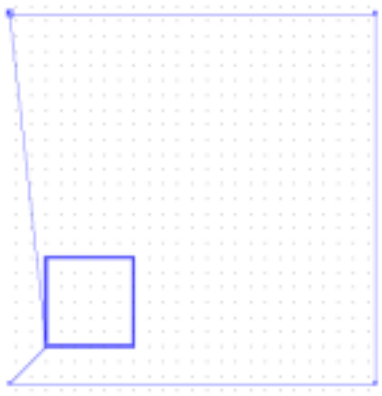


テスト 4

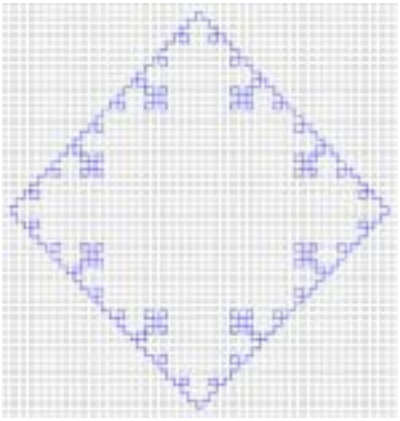


テスト 5

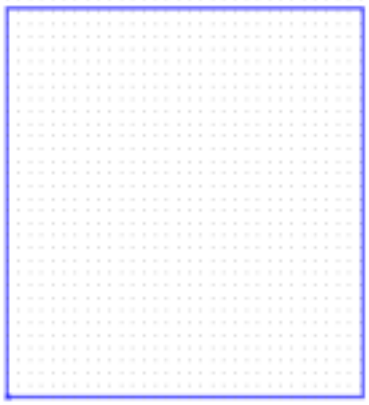
テスト 6



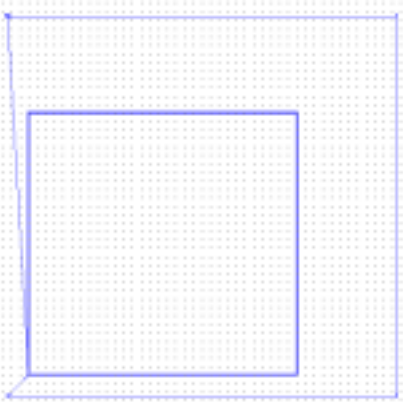
テスト 7



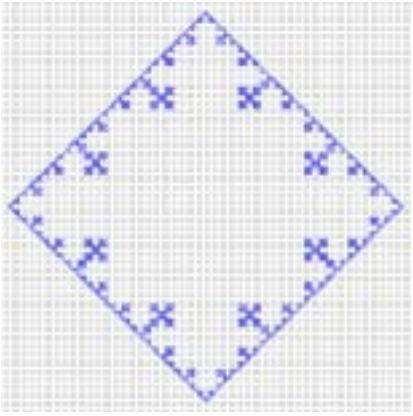
テスト 8



テスト 9

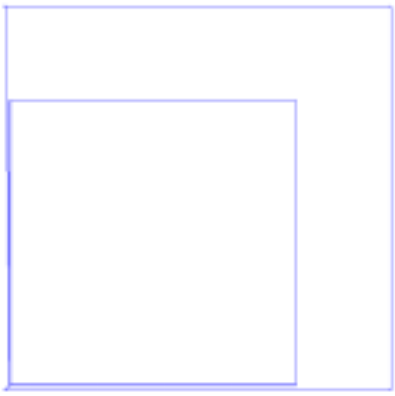


テスト 10



テスト 11

テスト 12



5.5. テスト結果

以下に、テスト結果の一覧表とグラフを、テスト毎に載せる。

●テスト 1　relate 命令　結果一覧表

relateテスト結果　　**(単位:秒)**　　［A：マシンA　B：マシンB］

テスト1	1.1	1.2	1.3	1.4	1.5	1.6	1.7
単独実行 A	0.1104	0.1011	0.0991	0.1082	0.1071	0.0582	0.0980
連続実行 A	0.0140	0.0180	0.0160	0.0200	0.0200	0.0050	0.0140
単独実行 B	0.9060	0.9160	0.9230	0.9520	0.9500	0.4960	0.8910
連続実行 B	0.1430	0.1820	0.1560	0.1870	0.2030	0.0440	0.1360

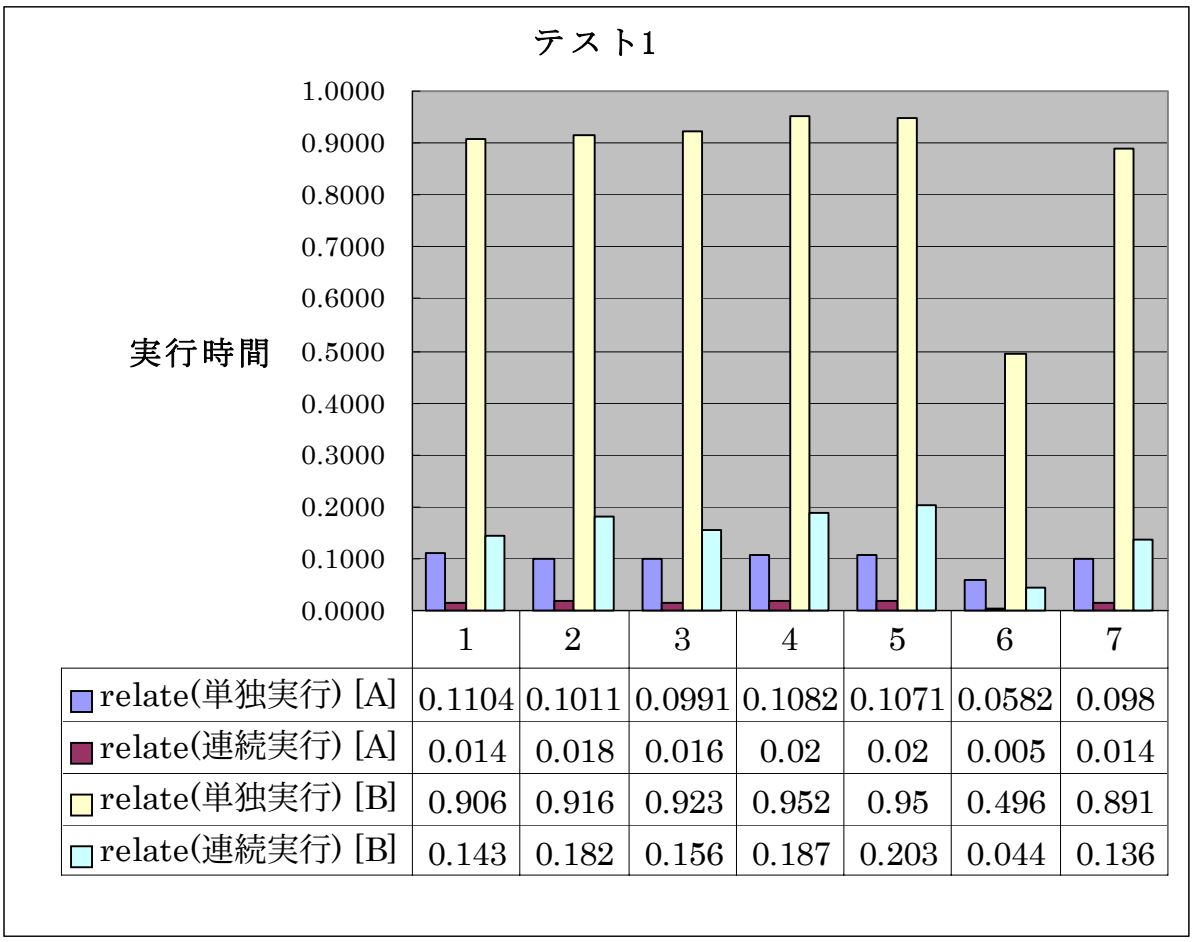
テスト2	2.1	2.2	2.3	2.4	2.5	2.6	2.7
単独実行 A	0.1157	0.1212	0.1223	0.1444	0.1514	0.0530	0.1121
連続実行 A	0.0220	0.0210	0.0200	0.0421	0.0370	0.0060	0.0180
単独実行 B	1.0640	1.0570	1.1030	1.2510	1.3150	0.5050	1.0560
連続実行 B	0.2020	0.2050	0.2320	0.3560	0.3460	0.0660	0.1740

テスト3	3.1	3.2	3.3	3.4	3.5	3.6	3.7
単独実行 A	0.1674	0.1590	0.1653	0.2243	0.2676	0.0540	0.1551
連続実行 A	0.0360	0.0362	0.0381	0.0771	0.0891	0.0050	0.0310
単独実行 B	1.2740	1.2520	1.3360	1.7920	1.9520	0.5230	1.2470
連続実行 B	0.3190	0.3490	0.3680	0.5380	0.7550	0.0610	0.2530

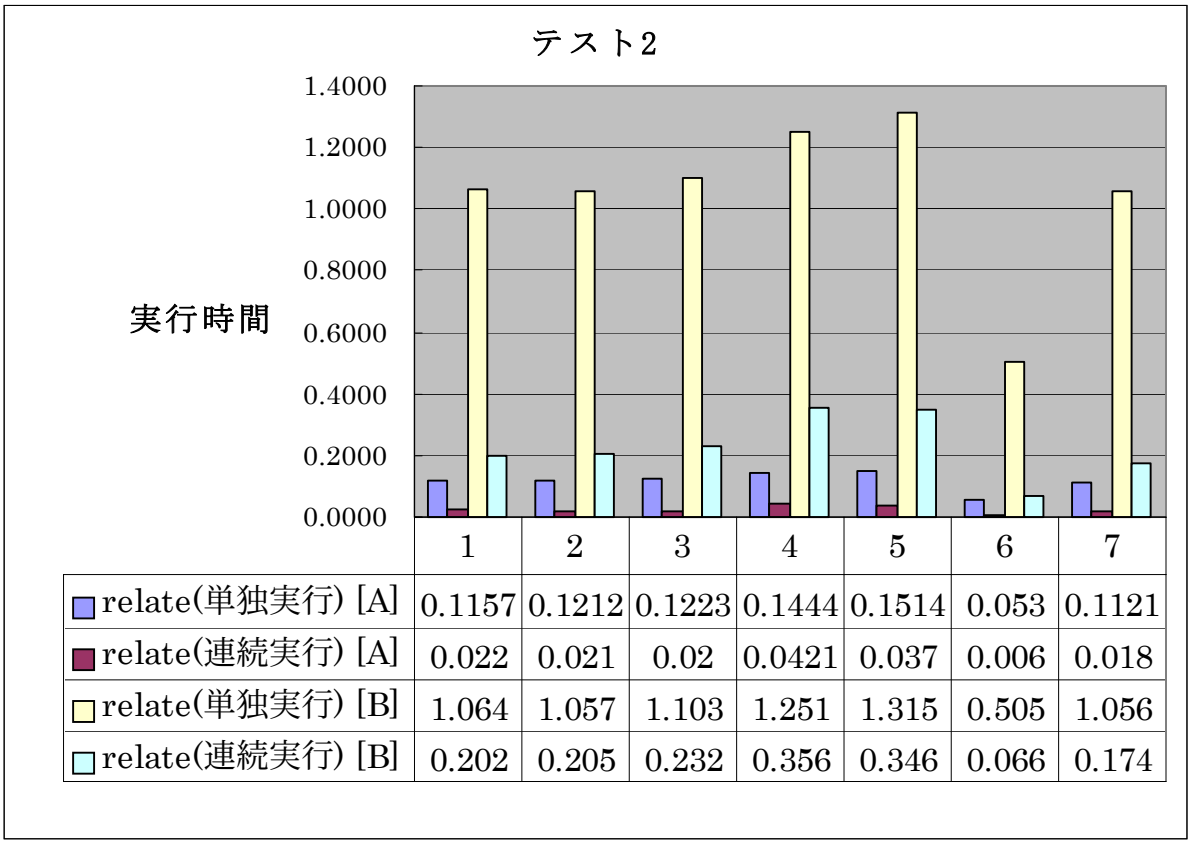
テスト4	4.1	4.2	4.3	4.4	4.5	4.6	4.7
単独実行 A	0.2614	0.2585	0.2517	0.3375	0.9194	0.0551	0.2443
連続実行 A	0.1231	0.1241	0.1091	0.1432	0.5549	0.0070	0.1051
単独実行 B	2.0210	2.0080	2.0420	2.7320	5.0520	0.5320	1.9660
連続実行 B	0.7920	0.7560	0.7300	1.1040	3.2420	0.0550	0.6710

テスト5	5.1	5.2	5.3	5.4	5.5	5.6	5.7
単独実行 A	0.7160	0.7140	0.5898	0.9534	6.4111	0.0591	0.5800
連続実行 A	0.5829	0.5839	0.4688	0.7851	6.0689	0.0100	0.4516
単独実行 B	3.5490	3.5160	3.0490	5.3080	33.6470	0.4890	2.8890
連続実行 B	2.8570	2.8770	2.3080	4.0450	31.7960	0.0650	2.3010

テスト 1　relate 命令　結果グラフ　　[A : マシン A　B : マシン B]

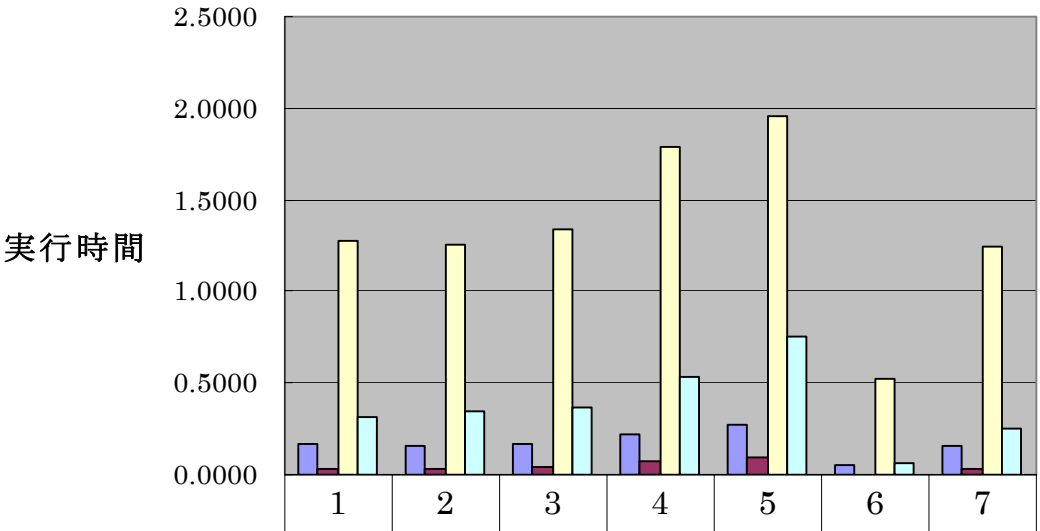


	1	2	3	4	5	6	7
relate(単独実行) [A]	0.1104	0.1011	0.0991	0.1082	0.1071	0.0582	0.098
relate(連続実行) [A]	0.014	0.018	0.016	0.02	0.02	0.005	0.014
relate(単独実行) [B]	0.906	0.916	0.923	0.952	0.95	0.496	0.891
relate(連続実行) [B]	0.143	0.182	0.156	0.187	0.203	0.044	0.136



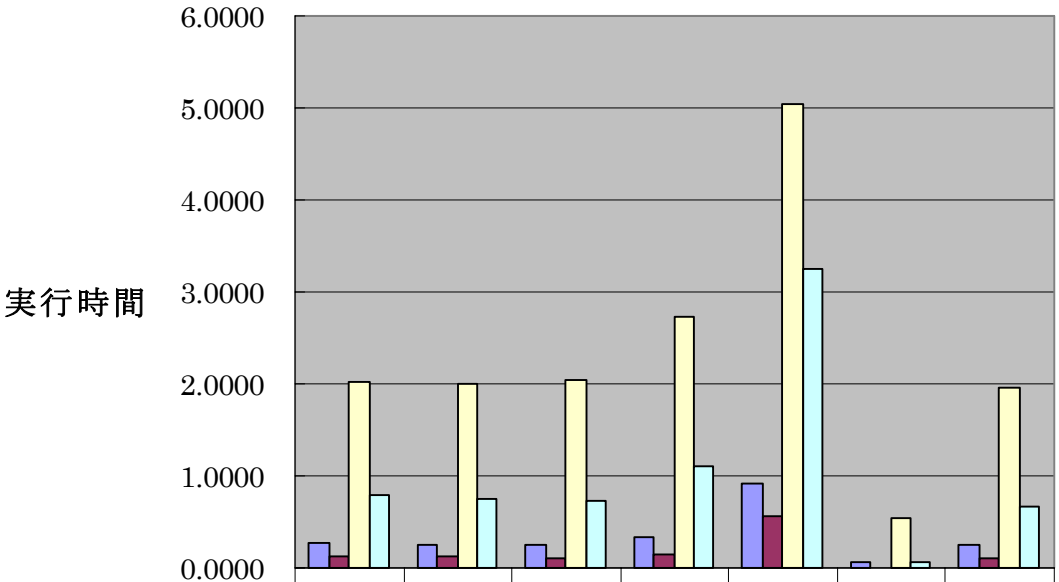
	1	2	3	4	5	6	7
relate(単独実行) [A]	0.1157	0.1212	0.1223	0.1444	0.1514	0.053	0.1121
relate(連続実行) [A]	0.022	0.021	0.02	0.0421	0.037	0.006	0.018
relate(単独実行) [B]	1.064	1.057	1.103	1.251	1.315	0.505	1.056
relate(連続実行) [B]	0.202	0.205	0.232	0.356	0.346	0.066	0.174

テスト3



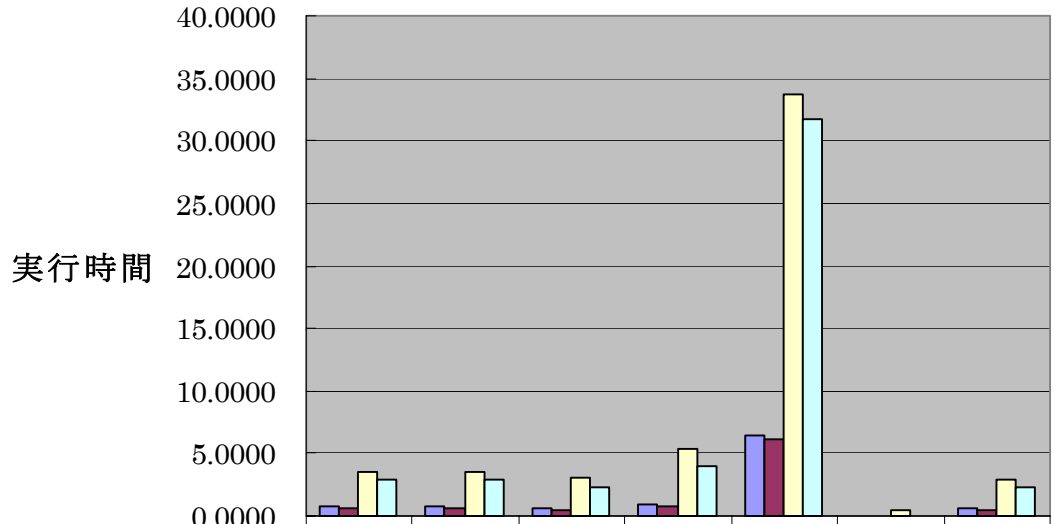
	1	2	3	4	5	6	7
relate(単独実行) [A]	0.1674	0.159	0.1653	0.2243	0.2676	0.054	0.1551
relate(連続実行) [A]	0.036	0.0362	0.0381	0.0771	0.0891	0.005	0.031
relate(単独実行) [B]	1.274	1.252	1.336	1.792	1.952	0.523	1.247
relate(連続実行) [B]	0.319	0.349	0.368	0.538	0.755	0.061	0.253

テスト4



	1	2	3	4	5	6	7
relate(単独実行) [A]	0.2614	0.2585	0.2517	0.3375	0.9194	0.0551	0.2443
relate(連続実行) [A]	0.1231	0.1241	0.1091	0.1432	0.5549	0.007	0.1051
relate(単独実行) [B]	2.021	2.008	2.042	2.732	5.052	0.532	1.966
relate(連続実行) [B]	0.792	0.756	0.73	1.104	3.242	0.055	0.671

テスト5



	1	2	3	4	5	6	7
relate(単独実行) [A]	0.716	0.714	0.5898	0.9534	6.4111	0.0591	0.58
relate(連続実行) [A]	0.5829	0.5839	0.4688	0.7851	6.0689	0.01	0.4516
relate(単独実行) [B]	3.549	3.516	3.049	5.308	33.647	0.489	2.889
relate(連続実行) [B]	2.857	2.877	2.308	4.045	31.796	0.065	2.301

●テスト 2　intersection 命令　結果一覧表

intersectionテスト結果　　（単位：秒）　　［A：マシンA　B：マシンB］

テスト1	1.1	1.2	1.3	1.4	1.5	1.6	1.7
単独実行 A	0.1101	0.1103	0.1114	0.1211	0.1213	0.1083	0.1103
連続実行 A	0.0190	0.0170	0.0170	0.0221	0.0220	0.0170	0.0161
単独実行 B	0.9780	0.9930	1.0010	1.1140	1.0540	0.9780	0.9730
連続実行 B	0.1800	0.1700	0.1990	0.2190	0.2360	0.1360	0.1490

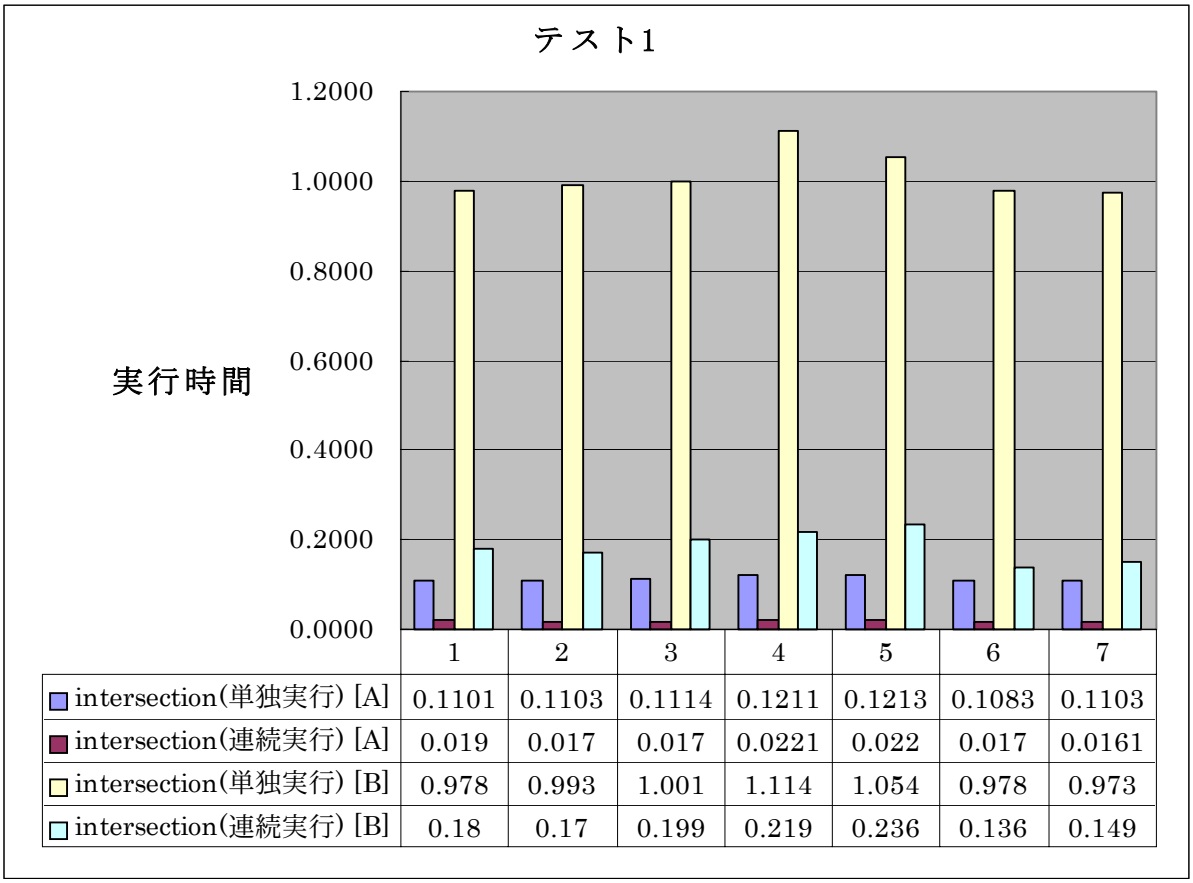
テスト2	2.1	2.2	2.3	2.4	2.5	2.6	2.7
単独実行 A	0.1290	0.1302	0.1322	0.1600	0.1731	0.1182	0.1203
連続実行 A	0.0221	0.0230	0.0262	0.0380	0.0380	0.0190	0.0210
単独実行 B	1.1140	1.1420	1.2190	1.3590	1.4720	1.0700	1.0750
連続実行 B	0.2080	0.2030	0.2540	0.4090	0.3680	0.1870	0.1980

テスト3	3.1	3.2	3.3	3.4	3.5	3.6	3.7
単独実行 A	0.1710	0.1711	0.1794	0.2724	0.3183	0.1590	0.1652
連続実行 A	0.0371	0.0370	0.0411	0.0842	0.1363	0.0311	0.0320
単独実行 B	1.3400	1.3490	1.4020	2.0590	2.2970	1.2870	1.3230
連続実行 B	0.3400	0.3000	0.3400	0.7350	1.0620	0.3190	0.3070

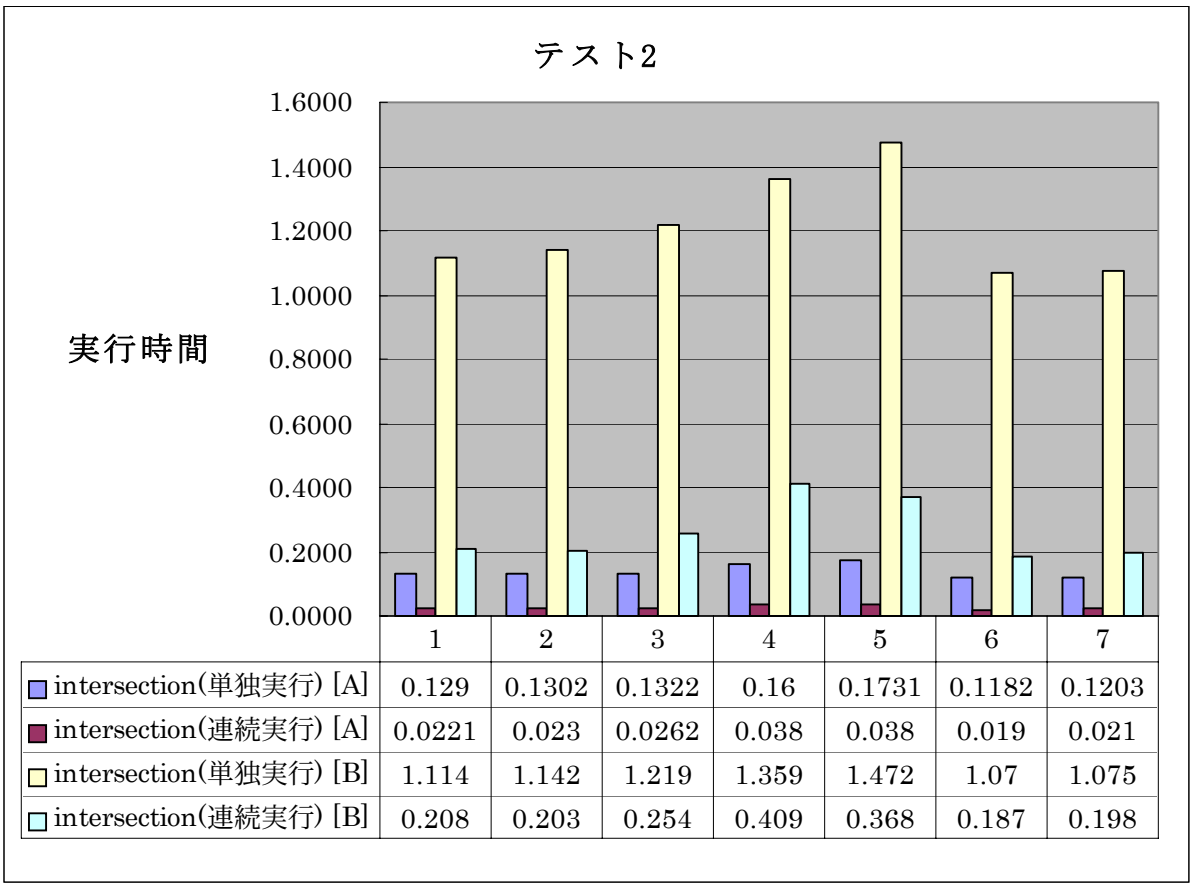
テスト4	4.1	4.2	4.3	4.4	4.5	4.6	4.7
単独実行 A	0.2653	0.2674	0.2865	0.5870	1.4961	0.2393	0.2484
連続実行 A	0.1252	0.1231	0.1101	0.2062	1.0975	0.0882	0.1044
単独実行 B	2.1050	2.1280	2.1600	3.2990	7.6620	1.9790	1.9720
連続実行 B	0.7740	0.7850	0.9220	1.5070	5.5810	0.6020	0.6640

テスト5	5.1	5.2	5.3	5.4	5.5	5.6	5.7
単独実行 A	0.7166	0.7130	0.6168	1.1644	18.7491	0.5378	0.5864
連続実行 A	0.5849	0.5777	0.4795	0.9081	18.7731	0.4188	0.4550
単独実行 B	3.5210	3.5030	3.1670	6.4840	66.9590	2.6820	2.8840
連続実行 B	2.8760	2.8560	2.4170	4.8070	64.0500	2.0670	2.2490

テスト 2　intersection 命令　結果グラフ　　[A : マシン A　B : マシン B]

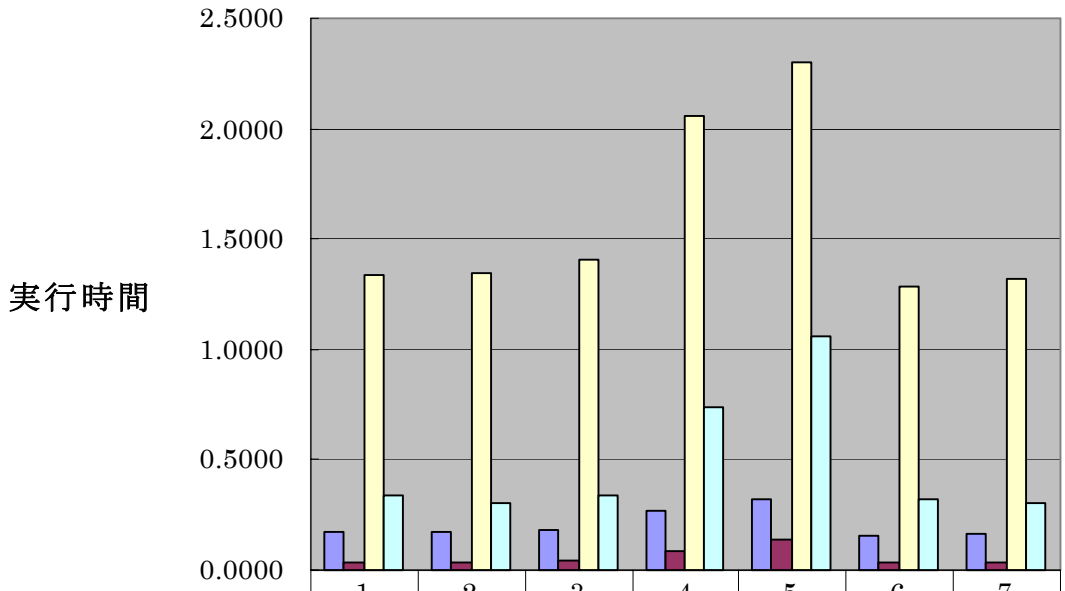


	1	2	3	4	5	6	7
intersection(単独実行) [A]	0.1101	0.1103	0.1114	0.1211	0.1213	0.1083	0.1103
intersection(連続実行) [A]	0.019	0.017	0.017	0.0221	0.022	0.017	0.0161
intersection(単独実行) [B]	0.978	0.993	1.001	1.114	1.054	0.978	0.973
intersection(連続実行) [B]	0.18	0.17	0.199	0.219	0.236	0.136	0.149



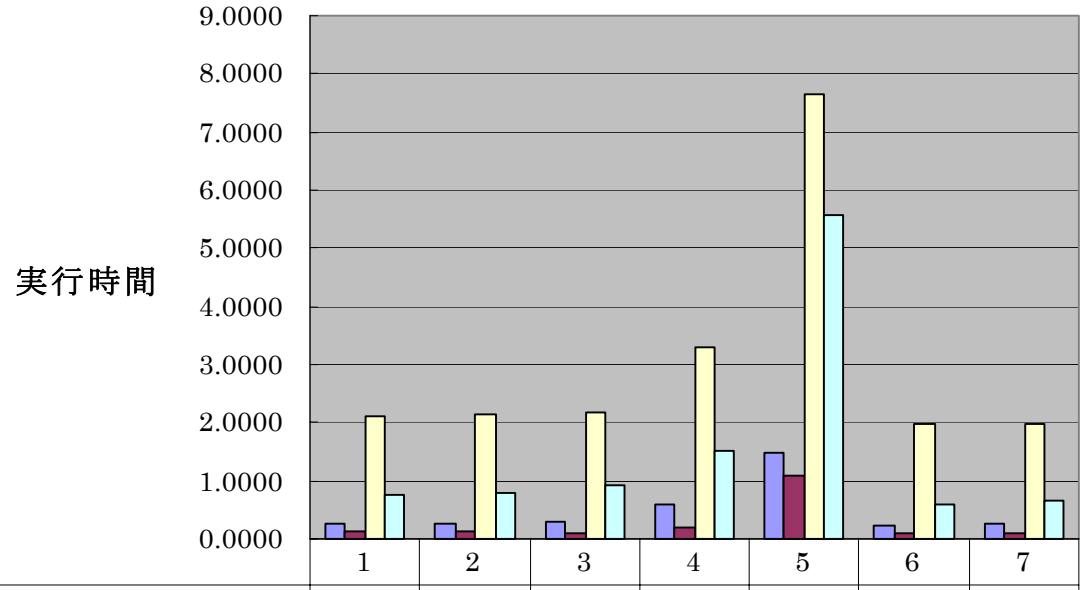
	1	2	3	4	5	6	7
intersection(単独実行) [A]	0.129	0.1302	0.1322	0.16	0.1731	0.1182	0.1203
intersection(連続実行) [A]	0.0221	0.023	0.0262	0.038	0.038	0.019	0.021
intersection(単独実行) [B]	1.114	1.142	1.219	1.359	1.472	1.07	1.075
intersection(連続実行) [B]	0.208	0.203	0.254	0.409	0.368	0.187	0.198

テスト3

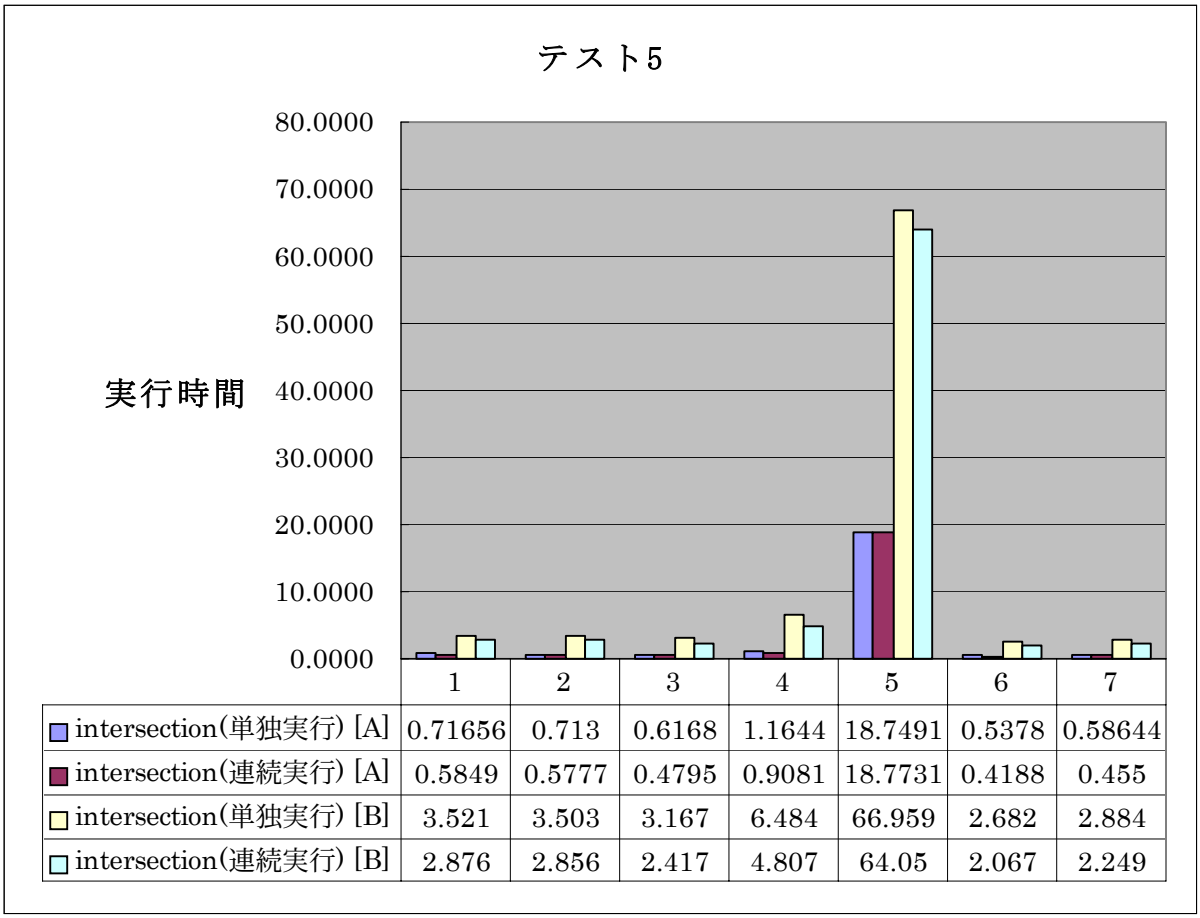


	1	2	3	4	5	6	7
intersection(単独実行) [A]	0.171	0.1711	0.1794	0.2724	0.3183	0.159	0.1652
intersection(連続実行) [A]	0.0371	0.037	0.0411	0.0842	0.1363	0.0311	0.032
intersection(単独実行) [B]	1.34	1.349	1.402	2.059	2.297	1.287	1.323
intersection(連続実行) [B]	0.34	0.3	0.34	0.735	1.062	0.319	0.307

テスト4



	1	2	3	4	5	6	7
intersection(単独実行) [A]	0.2653	0.2674	0.2865	0.587	1.4961	0.2393	0.2484
intersection(連続実行) [A]	0.1252	0.1231	0.1101	0.2062	1.0975	0.0882	0.1044
intersection(単独実行) [B]	2.105	2.128	2.16	3.299	7.662	1.979	1.972
intersection(連続実行) [B]	0.774	0.785	0.922	1.507	5.581	0.602	0.664



	1	2	3	4	5	6	7
intersection(単独実行) [A]	0.71656	0.713	0.6168	1.1644	18.7491	0.5378	0.58644
intersection(連続実行) [A]	0.5849	0.5777	0.4795	0.9081	18.7731	0.4188	0.455
intersection(単独実行) [B]	3.521	3.503	3.167	6.484	66.959	2.682	2.884
intersection(連続実行) [B]	2.876	2.856	2.417	4.807	64.05	2.067	2.249

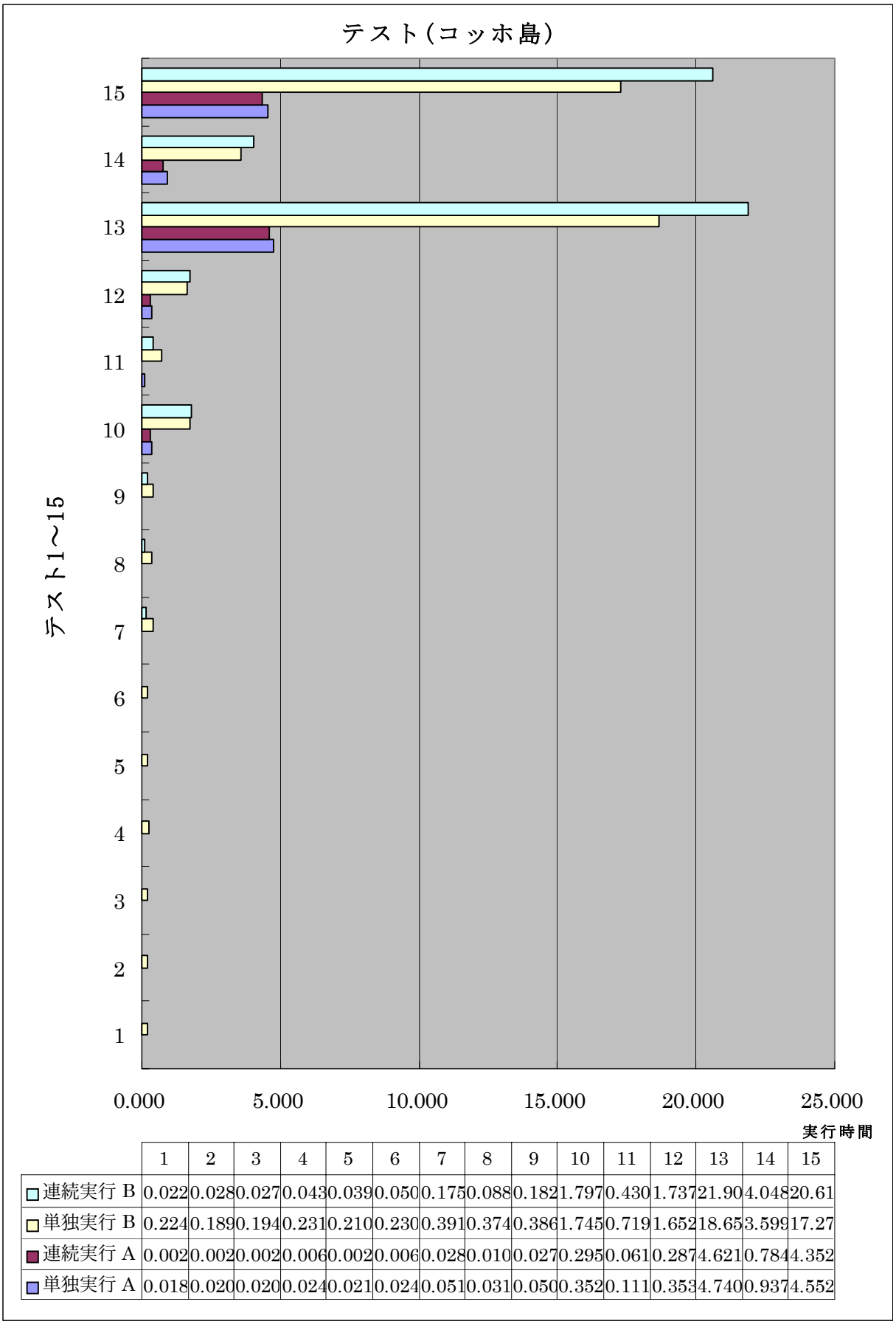
●テスト 3　convexhull 命令　結果一覧表

convexhullテスト結果　　（単位：秒）［A：マシンA　B：マシンB］

コッホ島	単独実行 A	連続実行 A	単独実行 B	連続実行 B
テスト1	0.0181	0.0020	0.2240	0.0220
テスト2	0.0200	0.0020	0.1890	0.0280
テスト3	0.0200	0.0020	0.1940	0.0270
テスト4	0.0240	0.0060	0.2310	0.0430
テスト5	0.0210	0.0020	0.2100	0.0390
テスト6	0.0242	0.0060	0.2300	0.0500
テスト7	0.0511	0.0280	0.3910	0.1750
テスト8	0.0311	0.0100	0.3740	0.0880
テスト9	0.0500	0.0270	0.3860	0.1820
テスト10	0.3525	0.2953	1.7450	1.7970
テスト11	0.1112	0.0610	0.7190	0.4300
テスト12	0.3536	0.2873	1.6520	1.7370
テスト13	4.7407	4.6217	18.6580	21.9030
テスト14	0.9372	0.7840	3.5990	4.0480
テスト15	4.5523	4.3524	17.2790	20.6190

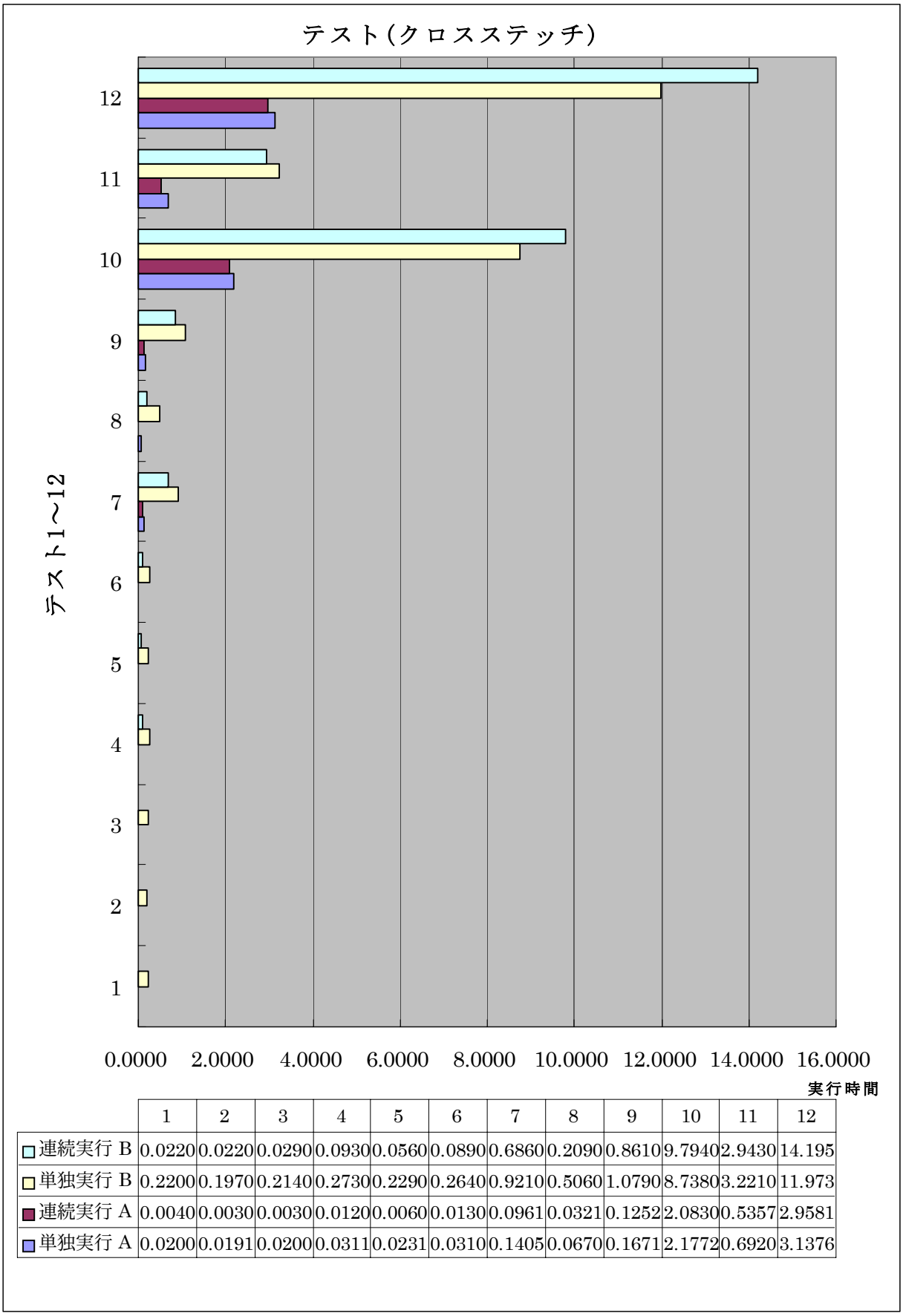
クロスステッチ	単独実行 A	連続実行 A	単独実行 B	連続実行 B
テスト1	0.0200	0.0040	0.2200	0.0220
テスト2	0.0191	0.0030	0.1970	0.0220
テスト3	0.0200	0.0030	0.2140	0.0290
テスト4	0.0311	0.0120	0.2730	0.0930
テスト5	0.0231	0.0060	0.2290	0.0560
テスト6	0.0310	0.0130	0.2640	0.0890
テスト7	0.1405	0.0961	0.9210	0.6860
テスト8	0.0670	0.0321	0.5060	0.2090
テスト9	0.1671	0.1252	1.0790	0.8610
テスト10	2.1772	2.0830	8.7380	9.7940
テスト11	0.6920	0.5357	3.2210	2.9430
テスト12	3.1376	2.9581	11.9730	14.1950

テスト 3　convexhull 命令　結果グラフ　 [コッホ島]　 [A : マシン A　B : マシン B]



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
連続実行 B	0.022	0.028	0.027	0.043	0.039	0.050	0.175	0.088	0.182	1.797	0.430	1.737	21.90	4.048	20.61
単独実行 B	0.224	0.189	0.194	0.231	0.210	0.230	0.391	0.374	0.386	1.745	0.719	1.652	18.65	3.599	17.27
連続実行 A	0.002	0.002	0.002	0.006	0.002	0.006	0.028	0.010	0.027	0.295	0.061	0.287	4.621	0.784	4.352
単独実行 A	0.018	0.020	0.020	0.024	0.021	0.024	0.051	0.031	0.050	0.352	0.111	0.353	4.740	0.937	4.552

[クロスステッチ]　[A：マシン A　B：マシン B]



	1	2	3	4	5	6	7	8	9	10	11	12
連続実行 B	0.0220	0.0220	0.0290	0.0930	0.0560	0.0890	0.6860	0.2090	0.8610	9.7940	2.9430	14.195
単独実行 B	0.2200	0.1970	0.2140	0.2730	0.2290	0.2640	0.9210	0.5060	1.0790	8.7380	3.2210	11.973
連続実行 A	0.0040	0.0030	0.0030	0.0120	0.0060	0.0130	0.0961	0.0321	0.1252	2.0830	0.5357	2.9581
単独実行 A	0.0200	0.0191	0.0200	0.0311	0.0231	0.0310	0.1405	0.0670	0.1671	2.1772	0.6920	3.1376

第６章　考察

6.1. テスト結果考察

(注　→ **6.2** とあるのは、6.2 節でさらに詳しく考察することを表す)

○テスト１　relate 命令

●頂点数が多いほど、実行時間は長くなる傾向がある。

…これは、頂点数が多いほど計算や判定に時間がかかるためである。

●単独実行より連続実行の方が実行時間は短い。

…これは、Java HotSpot などの性能向上化技術の成果であることは明白である。

表6.1.1. relate命令テスト1～5平均実行結果(左から実行時間の短い順（*x*.1と*x*.2は明白な順つかず))

relate	*x*.6	*x*.7	*x*.3	*x*.1	*x*.2	*x*.4	*x*.5
単独実行 A	0.0559	0.2379	0.2456	0.2742	0.2708	0.3536	1.5713
連続実行 A	0.5090	1.6098	1.6906	1.7628	1.7498	2.4070	8.5832
単独実行 B	0.0066	0.1239	0.1304	0.1556	0.1566	0.2135	1.3540
連続実行 B	0.0582	0.7070	0.7588	0.8626	0.8738	1.2460	7.2684

●表 6.1.1 より、テスト 1～5 全体でもっとも実行時間が短かったテストは *x*.6 であり、続いて *x*.7、*x*.3、*x*.1 か *x*.2、*x*.4、*x*.5 であった。

…これからまず、図形が完全に離れている場合(テスト 1.6、2.6、3.6、4.6、5.6)は、実行時間が大幅に短くなる傾向があることがわかった。全てのテストにおいてこのような傾向がみられ、特に頂点数が多い場合は傾向が顕著に現れた(テスト 5.5 と 5.6 では 2 桁も時間差が現れた)。→ **6.2**

次に、図形が接していない *x*.6、*x*.7 では、図形の接している *x*.1～*x*.5 よりも実行時間が大幅に短いことから、交差部分が無い場合はある場合よりも実行時間が短くなることがわかった。→ **6.2**

また、交差数が多い *x*.3 が、*x*.1、*x*.2 よりも実行時間が短かった。これは、図形の交差部分探索時に、図形交差数が *x*.1、*x*.2 より多いことから早い段階で図形の交差がわかり、DE-9IM の各値が時間的に比較的速く判定されるためであると思われる。また交差数が *x*.4 ほど多くないので、交差発見時の諸計算に時間があまりかからない。
逆に *x*.4 は、交差数が多く *x*.3 より交差発見時の諸計算の量が多いため、実行時間が *x*.3 より長い。

そして図形が完全一致している場合は、全てのテストにおいてもっとも実行時間が長くなった。図形交差部分が他の項目よりも圧倒的に多いため、交差発見時の諸計算、頂点や辺の走査に時間がかかったためと思われる。

○テスト 2　intersection 命令

●頂点数が多いほど、実行時間は長くなる傾向がある。

…これは、テスト 1 relate 命令と同様である。

●単独実行より連続実行の方の実行時間が短い。

…これは、テスト 1 relate 命令と同様である。

表6.1.2. intersection命令テスト1～5平均実行結果(左から実行時間の短い順 (*x*.1と*x*.2は明白な順つかず))

intersection	*x*.6	*x*.7	*x*.3	*x*.1	*x*.2	*x*.4	*x*.5
単独実行 A	0.2325	0.2461	0.2653	0.2784	0.2784	0.4610	4.1716
連続実行 A	1.5992	1.6454	1.7898	1.8116	1.8230	2.8630	15.8888
単独実行 B	0.0487	0.0556	0.0609	0.0657	0.0646	0.1113	0.4783
連続実行 B	0.6622	0.7134	0.8264	0.8756	0.8628	1.5354	14.2594

●表 6.1.2 より、テスト 1～5 全体でもっとも実行時間が短かったテストは *x*.6 であり、続いて *x*.7、*x*.3、*x*.1 か *x*.2、*x*.4、*x*.5 であった。

…基本的に、テスト 1 relate 命令と同様のケースが考えられる。→ **6.2**

●実行時間は、全体的にテスト 1 relate 命令よりも長くなった。

…いろいろ原因が考えられる。intersection 命令は結果を図形として返すが、relate 命令は結果を 9 文字の DE-9IM 文字列で返す。図形よりも文字列の方が作成に時間がかからない。その結果が実行時間の差に現れた、といえる。

　他の理由として、DE-9IM は正確な結果を数値として出さなくてよい(例えば、どこか一点だけでも交差部分が見つかれば、その図形全体が交差するといえる。その他の部分の交差チェックをする必要はなく、また交差部分の座標も必要ない)。一方、intersection 命令は、交差する部分が見つかれば、その座標を記憶し、続けて他の交差部分を探さなければならない。そのため、relate の方が実行時間は短くなる。

○テスト 3　convexhull

●頂点数が多いほど、実行時間は長くなる傾向がある。

…これは、テスト 1 relate 命令、テスト 2 intersection 命令と同様である。

●マシン A では、単独実行より連続実行の方が実行時間が短くなった。

…これは、テスト 1 relate 命令、テスト 2 intersection 命令と同様である。

表6.1.3. convexhull命令 コッホ島テスト1～15　平均実行結果(上から実行時間の短い順)

	単独実行 A	単独実行 B	連続実行 A	連続実行 B
正方形	0.2241	1.0182	0.1718	0.9266
大小正方形	1.0000	3.9482	0.9349	4.5230
コッホ島	1.0373	4.2498	0.9906	4.7880

表6.1.4. convexhull命令 クロスステッチテスト1～12　平均実行結果(上から実行時間の短い順)

	単独実行 A	単独実行 B	連続実行 A	連続実行 B
正方形	0.2003	1.0383	0.1442	0.8075
クロスステッチ	0.5922	2.5380	0.5488	2.6488
大小正方形	0.8389	3.3825	0.7748	3.7935

●表 6.1.3 より、コッホ島テストにおいてもっとも実行時間が短かった図形は正方形であり、次に大小正方形、そしてコッホ島であった。

一方、表 6.1.4 より、クロスステッチテストにおいてもっとも実行時間が短かった図形は正方形であり、次にクロスステッチ、そして大小正方形であった。→ **6.2**

●マシン B コッホ島テスト 10、12～15 およびクロスステッチテスト 10、12 では連続実行の方が実行時間が長くなった。

…まず、Java HotSpot に原因があったかどうかを考えてみる。Java HotSpot は、主として次のような機能を提供する。

・適応型コンパイラ
　…プログラムのクリティカルな部分(プログラムでもっとも時間がかかる部分)をコンパイルし、コンパイルしたコードを最適化する最善の方法を瞬時に判断する。

・メモリ割り当ておよびガベージコレクション
　…旧来のメモリ割り当て、ガベージコレクタより高速、効率的、高度な機能を実現

・スレッド同期
　…高速なスレッド処理機能を実現

Java HotSpot 技術は Java Virtual Machine 上で使用されるため、Java(SDK1.3)をインストールしているならば、特定のマシンでこの技術を利用できないことはない。また、オプションを何もつけないデフォルトの状態でプログラムを実行したため、マシン A も B も、連続実行時には Java HotSpot 技術がきちんと働いていたと考えられる。

次に、実行環境(マシン)に注目して考えてみる。すると、マシン A の同じテストでは、似たような結果が現れていない(マシン A では連続実行の方が単独実行よりも常に実行時間が短い)ことがわかる。このことから、マシン B の実行環境に何らかの原因があるのではないか、という仮説が立てられる。

マシン B はマシン A よりもハードウェア性能が大きく劣る。そこで、ハードウェア性能の点から考えると、次のようなことが思いついた。

・HotSpot は実行した部分をメモリに格納しておくため、HotSpot 未使用時よりもメモリを消費する
・Java 言語自体が実行にメモリを多く消費する。マシン B はメモリ容量がマシン A より少ない(64MByte)ため、利用可能な空きメモリは少なくなっている
・この結果が現れたテスト項目は、頂点数が多く、また途中の計算量が多い。
・自作プログラム makecoordiante のメモリ使用効率が悪い

これらのことから私は、このようなテスト結果がでた原因を次のように推測する。

テストの連続実行を続けるうちに、メモリの断片化が発生し、マシン B のメモリが足りなくなった。そのため、メモリのスワップ(メモリ容量が不足してきたら使われていないメモリ領域の内容を一時的にハードディスクに退避させ、必要に応じてメモリに書き戻す動作)が発生し、このスワップ時間分ほど実行時間が長くなってしまったと思われる。

なお、コッホ島テスト 11、クロスステッチテスト 11 でメモリスワップが発生しなかった原因としては、図形の形状によって処理時間に差がでるからではないかと思われる。この点については、6.2 節で詳しく述べる。 → **6.2**

6.2. JTS 解析結果からのテスト結果考察

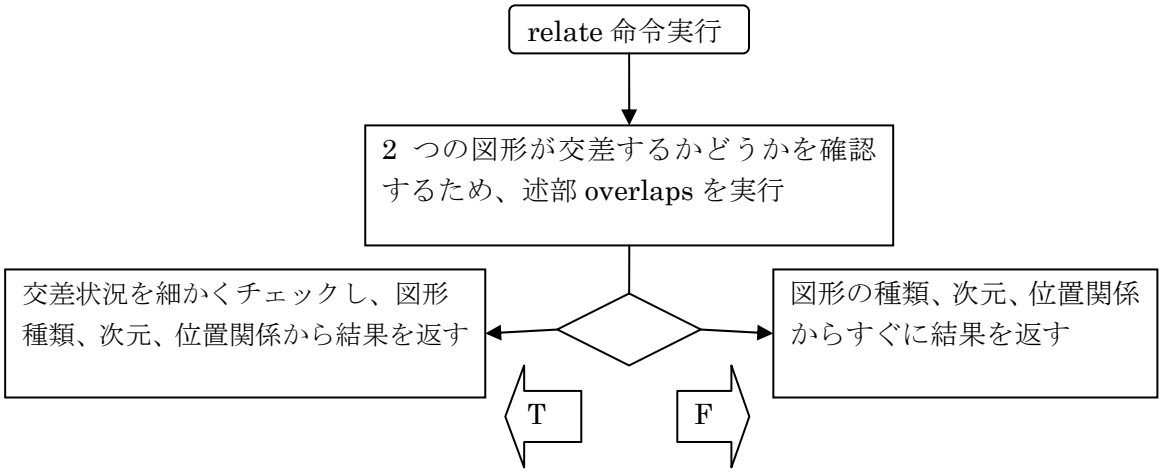
6.1 節においてテスト結果を考察したが、テスト結果のみを眺めていても、なぜこの場合で処理時間に差が出たのか、真の原因を理解することはできない。

そこで、JTS ソースプログラムおよびアルゴリズム解析結果から、もう一度テスト結果を考察してみることにする。

○relate 命令において図形が交差する場合としない場合で実行時間に差が出た原因

●relate 命令のフローチャート

relate 命令の主な処理のフローチャートは以下の通り。



●原因考察

relate 命令では、まず Binary Predicates の overlaps を実行し、2 つの図形の重なっている部分があるかをチェックする。2 つの図形が少しでも重なっているならば、図形の種類と位置関係、次元、交差状況など様々な項目を、図形の全ての頂点、辺に対してチェックする必要がある。そのため、実行時間が長くなってしまう。

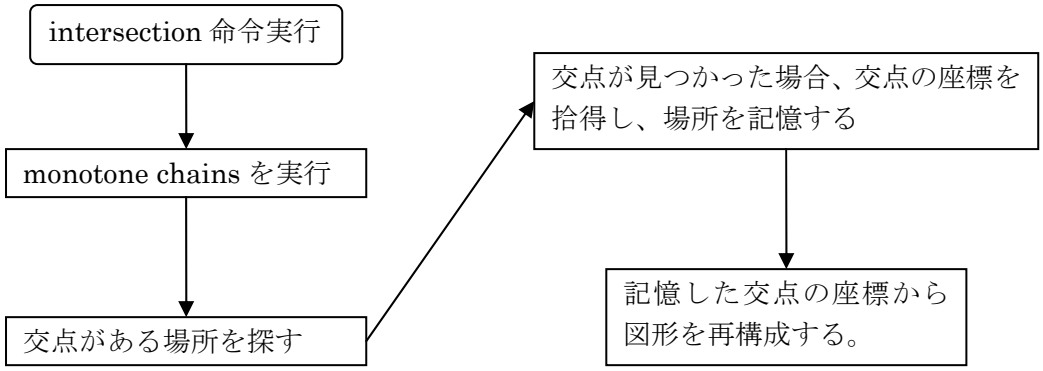
一方、2 つの図形が重なっていないならば、図形の位置関係は、完全に離れている場合もしくは一方が他方を含んでいる場合の 2 通りしかありえない。そのため、チェックする項目・処理時間が大幅に減り、結果がすぐに返ってくるのである。

実際、テスト結果をみると、特に頂点数が多いテストほど、図形が離れている場合とそうでない場合の時間差が大きくなっている。

○intersection 命令において図形が交差する場合としない場合で実行時間に差が出た原因

●intersection 命令のフローチャート

intersection 命令の主な処理のフローチャートは以下の通り。



●原因考察

intersection 命令は、2 つの図形の共通部分を探し、共通部分を新たな図形として返す命令である。

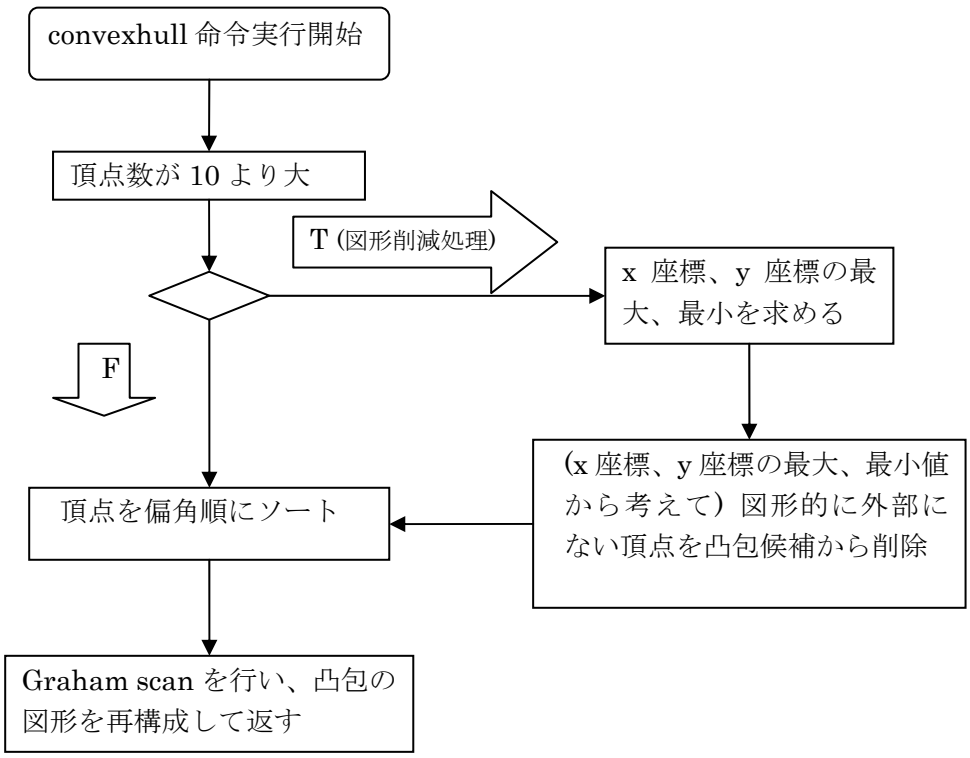
JTS では、まず 3.1 節において説明した Monotone chains を使用し、共通部分の高速探索を行う。交点が見つかると、その座標を記憶しておく。全ての交点を見つけ終わった後で、図形の位置関係や元の図形情報から、新しい図形を再構成する。

そのため、当然交点が少ない方が処理に時間がかからない。実際、テスト結果では、特に頂点数が多い場合は、この交点検索と図形再構成の時間の影響が顕著に現れている。

○convexhull 命令において図形種類により実行時間に差が出た原因

●convexhull 命令のフローチャート

convexhull 命令は、JTS ソースプログラム com.vividsolutions.jts.algorithm.ConvexHull において実現されている。このプログラムの主な処理のフローチャートは以下の通り。



●原因考察

convexhull のアルゴリズムは 3.3 節にて述べたが、JTS ではソートの前に、凸包にならない頂点の削除を行おうとしている。頂点数が 10 より大の場合、凸包の頂点にならない点を削除しておいてから、ソートを行う。凸包になる可能性のある頂点は、現在の最大・最小 x、y 座標より**図形的に Exterior(外部)**にある。よって、境界や内部にある頂点は削除しても、結果に影響はない。また、削減の計算量はせいぜい $O(N)$ である。ソートの計算量は $O(N\log N)$ であるため、ソート前に不要な頂点を削除しておくことは非常に有益である。

テストした図形は全て頂点数 10 より大であったため、テストデータは全て頂点削減処理を行っている。その中でも正方形は、4 隅の頂点以外は図形的に境界上に存在していたため、上記の頂点削減処理で 4 隅の頂点以外が全て削除される。したがって、ソートと Graham scan の処理時間は実質 0 であり、実行時間が極端に短くなったのである。

さて、コッホ島と大小正方形の比較では、コッホ島の方が処理時間は長かった。一方、クロスステッチと大小正方形の比較では、大小正方形の方が処理時間は長かった。次に、この点について考えてみたい。

大小正方形は、頂点削減処理において頂点が一つも削除されない(外側の 4 隅の頂点以外は全て図形的に外部に存在しているため)。また、コッホ島・クロスステッチも頂点が削除されない(最大、最小 x、y 座標がもっとも図形の外側にあるため)。よって、この 2 つの図形の処理時間の差は、ソートあるいは Graham scan 実行時に現れることになる。

大小正方形は頂点が順序よく並んでおり、ソートに時間はかからない。一方で、Graham scan 時に凸包となる頂点は外側の 4 カ所以外現れない。そのため、Graham scan 時に無駄な走査が行われ、実行時間がかかる。

よってこの図形は、ソートで時間はかからないが Graham scan で実行時間がかかってしまう。しかしソートの方が処理時間はかかるため、この図形の実行時間は比較的早いほうである。

次にコッホ島・クロスステッチについて考えるが、次の図は、コッホ島、クロスステッチの凸包図形を拡大したものである。

図 6.2.1. 4 次コッホ島凸包の拡大図（黄色領域が凸包領域　ただし頂点・ベクトル方向は描写せず)

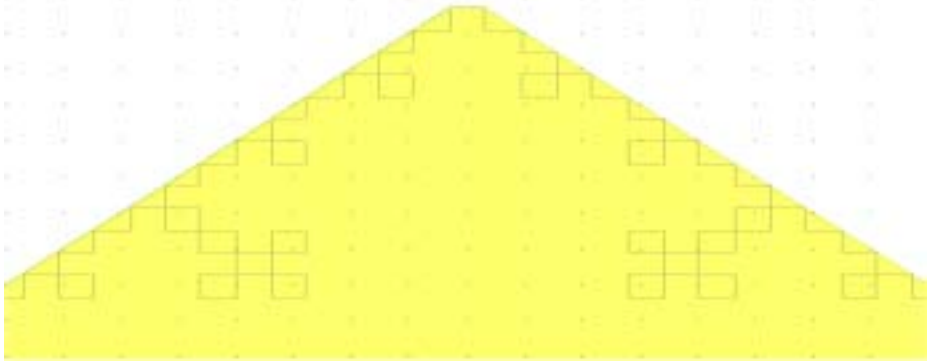


図 6.2.2. 3 次クロスステッチ凸包の拡大図(黄色領域が凸包領域　ただし頂点・ベクトル方向は描写せず)

これからわかることは、凸包となる頂点はクロスステッチの方が多いことである。裏を返せば、コッホ島の方が凸包と関係ない不要な頂点が多いことになる。凸包でない頂点が多いと、Graham scan 時の走査に時間がかかる。一方、ソート時間は両図形とも大小正方形より時間がかかる。

これらのことから、テスト結果のようになった原因は、主に Graham scan の処理時間の差によるものと推測される。

ソート、Graham scan ともに処理時間がかかるコッホ島はもっとも実行時間が長くなり、ソートに処理時間がかからないが Graham scan で処理時間がかかる大小正方形はコッホ島より実行時間が短くなる。しかし、ソートに処理時間がかかるが Graham scan の処理時間が短くなるクロスステッチは、大小正方形より実行時間が短くなるのである。

6.3. JTS 改良点

JTS 解析中に、ソースプログラムにおいて改良すべき点が見つかったので、ここで述べておく。

○Monotone chains の領域分割について

図形を Monotone chains 領域に分ける際に参照する、JTS ソースプログラム Quadrant において、改良すべきと思われる部分を発見した。

この Quadrant では、Monotone chains 領域であるかどうかを、図 6.3.1 の四象限を使用して判定している。ここで使用している理論は、次の通りである。

もし図形が Monotone chains 領域を持つなら、i+1 番目の頂点座標から i 番目の頂点座標を減算した座標値の象限は、常に同じ象限である （図形の最初の頂点 ≦ i < (図形の最後の頂点-1)）

JTS では、次の方法で象限を分けている。

Ⅰ．(i+1 番目の x 座標値 - i 番目の x 座標値) = (0 以上)ならばⅡへ。そうでないならⅢへ。
Ⅱ．(i+1 番目の y 座標値 - i 番目の y 座標値) = (0 以上)ならば、第 0 象限である
そうでないならば、第 3 象限である。
Ⅲ．(i+1 番目の y 座標値 - i 番目の y 座標値) = (0 以上)ならば、第 1 象限
そうでないならば、第 2 象限である。

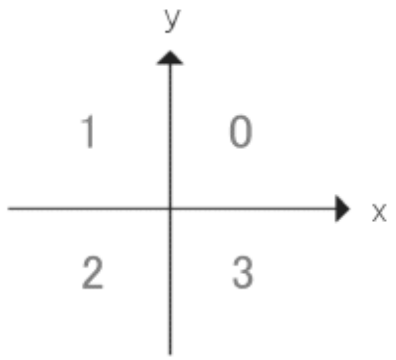


図 6.3.1. Quadrant における四象限の位置づけ

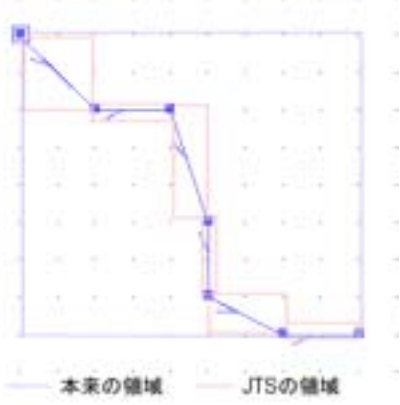


図 6.3.2. 領域分割結果の違い

例えば、図形 LineString の座標値が(100,100),(150,150),(180,200),(300,300)ならば、((i+1 番目の x、y 座標値) – (i 番目の x、y 座標値)) = (0 以上,0 以上)より、全ての頂点は第 0 象限である。よって、この図形は Monotone chains であるといえる。

一方、図形 LineString の座標値が(100,100),(150,150),(100,50),(300,300)ならば、((3 番目の x、y 座標値) – (2 番目の x、y 座標値)) = (-50,-100)より、第 3 象限となる。その他の頂点は第 0 象限である。つまり、常に一定方向を向いていない図形であるため、Monotone chains 領域ではない。

ここで問題となるのが、下線部分「0 以上」という条件分けである。x 軸または y 軸に対しての垂線は決して自己交差しないので、Monotone chains 領域に含まれるはずである。しかしそのような垂線は、JTS の方法では、(i+1 番目の x(y)座標値 – i 番目の x(y)座標値) = 0 になってしまう。つまり、垂線を含む図形は、i 番目に第 j 象限にある場合でも、j と異なる第 k 象限に象限を変更され、Monotone chains 領域が本来の領域より小さくなってしまうのである。(例 図 6.3.2)

この結果、無駄に Monotone chains 領域が増え、計算量および実行時間が増加してしまうのである。

おそらく、この程度の Monotone chains 領域の増加は処理時間に影響を与えない、と考えられているため、このような方法をとったのであろうが、この点について直せば、若干ながら効率が上がるのではないかと思われる。

○ConvexHull のソートについて

ConvexHull のソースプログラムを解析中に、改良すべき部分を発見したので述べておく。

JTS では、凸包のソートにおいて、クイックソートなど $O(N\log N)$ のアルゴリズムを使用せず、バブルソート($O(N^2)$)を使用している。おそらく、ソート前に頂点削除アルゴリズムを実行するため、開発者がバブルソートでも十分な結果が得られると判断したためと思われる。

ここをクイックソートなどの $O(N\log N)$ のソートに変更すれば、さらに良い結果が得られるであろう。

第７章　まとめ

○JTS に対する総合的評価

本研究で、テスト、JTS 機能・アルゴリズムの解析を行うことで、JTS が持つ様々な機能を評価することができた。ここでは、本論文で述べてきたことをまとめ、総合的な JTS 評価を行う。

まずテスト結果であるが、テストを実行して体感したことは、JTS の処理時間の速さであった。現在の標準性能から大きく下回るマシン B ですら、想像以上の速さで結果が帰ってきたことには驚いた。テスト前は、実行が遅くマシンパワーを必要とするという Java 言語の仕様から、マシン A はともかくマシン B は結果が返ってくるまで数分かかるのではないか、と考えていた。しかし、テスト結果からもわかるように、最悪でもマシン B は約 1 分で結果を返している。このことから、実行結果のみから判断しても、JTS の性能の高さを感じ取ることができた。

また、Java の最適化技術を使用した結果、さらに結果が速く得られることが結果から読みとれた。しかし、テストの一部で最適化が逆効果になっている場合もあった。この原因はハードウェア性能によるものと判断したが、これより、ある程度のハードウェア性能がなければ効果が得られないことが判明した。私の自作プログラムの性能が悪い、ということも原因の 1 つと考えられるが、JTS をシステムに組み込む場合には、この点に注意すべきかもしれない。

次に、テスト結果を元にソースプログラムを解析した結果であるが、プログラムが整理されていて理解しやすく、コーディングレベルの高さを感じた。また、実装されている技術・アルゴリズムは、いずれも簡潔かつ高性能のものであり、解析の結果、条件により様々な枝刈りを行って処理時間を短縮していることが読みとれた。

一方で、本研究で扱った JTS が version0.9 と完成前であったためか、ソートや交差状況チェック部分において改良すべき部分を発見した。今回は時間の都合により改良できなかったが、改良後、処理時間がどのように変化するかをぜひ確かめたかった。

実際の GIS システムでは、図形同士が交差しているかどうかを判定する、または検索条件に一致する共通部分を抽出する、という操作がもっともよく使用されるはずである。このことから、図形が交差していない場合に高速で結果が返ってくる JTS の仕組みは、大変実用的であると思う。また、頂点数の多い図形よりも、形状の複雑な図形のテストの方が実際のテスト対象頻度は多いはずであり、フラクタル図形のような複雑な図形でもよい実行時間を返すような JTS は、極めて現実的に有効ではないかと考える。

これらのことから私は、JTS は十分な性能をもつ、優れた API であると判断する。JTS の利用方法として、単独で使用する事も考えられるが、他のソフトウェア、特に ORDB と連携させて、ORDB で最適化しきれない部分を補うことで、性能を最大限に引き出すことができると考える。

○本研究を終えての感想

残念であったことは、本研究で扱った JTS は完成前の version 0.9 であったことである。JTS の全機能が完全な形で使用できないため、すべての命令についてテストできなかった。また JTS ドキュメントも未完成であり、より詳しく JTS を理解することはできなかった。2002 年 2 月 2 日、version 1.0 が使用できることを確認したが、これもまだ完成版ではないようである。まだ JTS は開発途上であるとはいえ、研究の完成度を高めるためにも、完成版を扱いたかった。

JTS 開発の遅れから、入手時期が当初の予定より大幅に遅れてしまったことも、本研究の大きな妨げとなった。JTS を入手できたのは当初の予定から 2 ヶ月も遅れてであり、JTS ソースプログラム解析・実装・テストに十分な時間が確保できなかった。またテストに関しても、全ての図形のパターンに対してテストしたとはいえないため、さらなるテストが必要かもしれない。

だが、JTS 入手が遅れた不幸中の幸いとして、自作の図形プログラム makecoordinate をじっくり改良することができたことが挙げられる。様々な機能を追加することで、テストデータの作成・編集が楽になり、1 頂点で図形が交差する場合のテストも無事に行うことができた。まだまだ手を加えたい部分はあるが、この makecoordinate は十分な性能をもつ図形作成・編集ツールである、と私は自信を持って言える。

○今後の課題

本研究の今後の課題としては、完成した JTS の内容の解析、JTS 全機能の実装、テスト項目・テスト図形の追加、そして JTS に対して独自機能を追加し、オリジナルの JTS と性能を比較することが挙げられる。

また応用として、JTS と ORDB とを連携したシステムを構築し、性能を評価することが挙げられる。

参考 URL

Vivid Solutions Inc.　JTS Developing Projects
http://www.vividsolutions.com/jts/jtshome.htm

参考文献 (著者 作品名 出版社 総ページ数 出版年の順に記載(2)は Java Docment のため総ページ数無し))

1) Open GIS Consortium,Inc.
"OpenGIS Simple Features Specification For SQL Revision1.1"
OpenGIS Project Document 99-049., 78pp, 1999

2) VIVID SOLUTIONS Inc.
JavaDoc Java Topology Suite API
VIVID SOLUTIONS Inc., 2001

3) VIVID SOLUTIONS Inc.
JAVA TOPOLOGY SUITE Technical Specifications
VIVID SOLUTIONS Inc., 32pp, 2001

4) VIVID SOLUTIONS Inc.
JTS Technical Specifications
VIVID SOLUTIONS Inc., 29pp, 2001

5) VIVID SOLUTIONS Inc.
JTS TestBuilder & TestRunner User Guide
VIVID SOLUTIONS Inc., 20pp, 2001

6) 杉原厚吉
「計算幾何工学」
培風館, 231pp, 1994 年

7) スティーブン・ホルツナ, 武藤健志 監修, トップスタジオ 訳
「BlackBook Java プログラミング」
インプレス, 771pp, 2000 年

8) 西村哲哉
「自己相似フラクタル図形を用いた ORDB 空間データベンチマークテスト」
応用情報学講座 田中研究室 卒業研究, 54pp, 2001 年

9) プレパラータ/シェーモス、浅野孝夫/浅野哲夫 訳
「計算幾何学入門」
総研出版, 443pp, 1992 年

10) M.ドバーグ/M.ファン.クリベルド/M.オーバマーズ/O.シュワルツコップ
「コンピュータ・ジオメトリ 計算幾何学：アルゴリズムと応用」
浅野哲夫 訳, 近代科学社, 441pp, 2000 年

謝辞

本研究において様々なご指導、助言をしていただきました田中章司郎教授に心から感謝いたします。

またお忙しい中、JTS の提供と数々の助言をしていただきました、Martin Davis 氏をはじめとする Vivid Solutions 社の方々には、この場をかりて厚くお礼を申し上げます。

そして、本研究室所属の江野本さん、貫目さん、辰巳さん、平田さん、藤井さん、森下さん、森本さん、荒木君、驛君、高木君、富岡君、中村君には、本研究に対して様々な助言、サポートをしていただきました。深く感謝いたします。