The ACP Library

Elisha Sacks
ACP is a C++ library for robust computational geometry.
You will use ACP for the programming assignments.
I will post ACP programs for most algorithms that we study.
ACP supports parameters, datums, objects, and predicates.
Parameters

- The Parameter class represents real parameters.
- Input parameters are initialized with floating point numbers.
- Derived parameter are constructed from prior parameters: $a + b, -a, a - b, a \times b, a.rcp(), a/b, a.sqrt()$.
- Either argument of a binary operator can be a number.
- The sign of a parameter is given by the sign() method.
- Parameters occur solely in datums, objects, and predicates.
Datums

- A datum is a templated class that represents a vector of parameters.
- The datums PV2 and PV3 represent 2D and 3D vectors.
- The standard vector operations are supported: $a + b$, $2 \times a$, $a\.dot(b)$, $a\.cross(b)$, $a\.length()$, $a\.unit()$.
- Users can define other datums, such as circles.
Objects

- The Object class is templated by a datum.
- Users define classes of objects as subclasses of Object.
- Examples: Point is a subclass of Object<PV2> and CircleCenter is a subclass of Point.
- Objects are input or constructed.
- An input object initializes its datum with numbers.
- Optionally, these values are perturbed: \( a \) is replaced by \( a(1 + r) \) with \( r \) uniform in \([-10^{-8}, 10^{-8}]\).
```cpp
class Point : public Object<PV2> {
public:
  Point (double x, double y, bool perturb = true) : Object(PV2<double>(x, y), perturb) {}

  void print (bool newline = true) {
    PV2<double> p = getApproxMid();
    cerr << "(" << p.x << " - " << p.y << ")";
    if (newline) cerr << endl;
  }
};
```

- Point is an input object class.
- The input values x and y are perturbed by default.
- Replacing “true” by “false” omits the perturbation.
- The print method is for program output only.
A constructed object stores pointers to antecedent objects.
Its datum is computed from those of its antecedents.
The code is specified by the DeclareCalculate macro.
It is templated by type $N$.
The datum of an antecedent object is accessed with its $\text{get} \langle N \rangle ()$ method.
Example: the intersection point of line segments $ab$ and $cd$. 

**Constructed Objects**
class LineIntersection : public Point {
    Point *a, *b, *c, *d;

    // DeclareCalculate (PV2)
    PV2<N> aa = a->get<N>(), cc = c->get<N>(),
    u = b->get<N>() - aa, v = d->get<N>() - cc;
    N k = (cc - aa).cross(v)/u.cross(v);
    return aa + k*u;
}

public:
    LineIntersection (Point *a, Point *b, Point *c,
    Point *d) : a(a), b(b), c(c), d(d) {}
};
Predicates

- A predicate is a subclass of the Primitive class.
- Its value is computed from its antecedent objects.
- The DeclareSign macro computes a parameter.
- The sign of this parameter is returned.
- The sign is computed probabilistically with the Sacks Milenkovic algorithm.
- It is nonzero for perturbed input, except for identities.
- Example: the left turn predicate for points $a$, $b$, and $c$. 
Left Turn Predicate

class LeftTurn : public Primitive {
    Point *a, *b, *c;

    DeclareSign {
        PV2<N> aa = a->get<N>(), bb = b->get<N>(),
        cc = c->get<N>();
        return (cc - bb).cross(aa - bb);
    }

    public:
    LeftTurn (Point *a, Point *b, Point *c) :
        a(a), b(b), c(c) {}
};

    // sample call
    Point *a = ..., *b = ..., *c = ...
    int s = LeftTurn(a, b, c);
Point x Order Predicate

class XOrder : public Primitive {
    Point *a, *b;

    DeclareSign {
        return b->get<N>() .x - a->get<N>() .x;
    }

public:
    XOrder (Point *a, Point *b) : a(a), b(b) {};

    ▶ This predicate is used in the generic convex hull algorithm.
    ▶ How is lexicographic order implemented?
main () {
    acp::enable();
    double ax, ay, bx, by, cx, cy, dx, dy;
    cin >> ax >> ay >> bx >> by  
       >> cx >> cy >> dx >> dy;
    PTR<Point> a = new Point(ax, ay),
              b = new Point(bx, by), c = new Point(cx, cy),
              d = new Point(dx, dy);
    if (intersects(a, b, c, d)) {
        PTR<Point> p = new LineIntersection(a, b, c, d)
        cerr << "Intersection point is";
        p->print();
    }
    else
        cerr << "No intersection." << endl;
    acp::disable(); }
void convexHull (Points &p, Points &h) {
    for (int i = 1; i < p.size(); ++i)
        if (YOrder(p[i], p[0]) == 1)
            swap(p[i], p[0]);
    sort(p.begin() + 1, p.end(), CCWOrder(p[0]));
    int m = 0;
    for (int i = 0; i < p.size(); ++i) {
        h.push_back(p[i]);
        ++m;
        while (m > 2 &&
            LeftTurn(h[m-3], h[m-2], h[m-1]) == -1)
            h[m-2] = h[m-1];
        h.pop_back();
        --m;
    }
}
class CCWOrder {
    Point *o;

public:
    CCWOrder (Point *o) : o(o) {} 
    bool operator()(Point *a, Point *b) const {
        return LeftTurn(a, o, b) == -1;
    }
};