ACP Parameters

- Parameter class for real parameters.
- Perturbed input parameter: Parameter::input(3).
- Derived parameters: $a + b, a - b, a \times b, a/b, a.sqrt()$.
- Numbers (unperturbed): $a + 1, 3.14 \times a, (a + b)/2$.
- The sign of a parameter is given by the sign() method.
- Parameter vectors PV2 and PV3 with standard operations: $a + b, 2 \times a, a.dot(b), a.cross(b)$.
- Parameters only occur in calculate methods and in primitives.

ACP Objects

- Object class template for geometric objects: parameters, constituent objects, method that calculates parameters.
- Point: object whose coordinates are a PV2.
- InputPoint: extends Point with coordinate values.
- Vector: extends Point with constituent points t and h and calculate method $h - t$.

ACP Primitives

- Primitive class template for predicates.
- Arguments are objects and output is a sign.
- The sign is computed using interval arithmetic.
- If ambiguous, precision is increased as necessary using MPFR.
- XOrder primitive on points a and b returns the sign of $b - a$.
- LeftTurn primitive on a, b, c returns the sign of $\mathrm{circ}(a, b, c)$.

point.h

```
#include "object.h"
using namespace acp;

class Point : public Object<PV2> {};

typedef vector<PTR<Point>> Points;

Primitive2(XOrder, Point*, a, Point*, b);
Primitive2(CCW, Point*, a, Point*, b);
Primitive3(LeftTurn, Point*, a, Point*, b,
           Point*, c);
```

point.C

```
#include "point.h"

int XOrder::sign () {
  return (b->get().x - a->get().x).sign();
}

int CCW::sign () {
  return a->get().cross(b->get()).sign();
}

int LeftTurn::sign () {
  PV2 u = c->get() - b->get(),
      v = a->get() - b->get();
  return u.cross(v).sign();
}
```

point.h continued

```
class InputPoint : public Point {
 public:
  InputPoint (double x, double y) {
    set(PV2::input(x, y));
  }
};

class Vector : public Point {
  PV2 calculate () {
    return h->get() - t->get();
  }
 protected:
  PTR<Point> t, h;
 public:
  Vector (Point *t, Point *h) : t(t), h(h) {}
};
```

point.h continued

```
class LineIntersection : public Point {
  PV2 calculate () {
    PV2 u = b - a, v = d - c;
    Parameter k = (c - a).cross(v)/u.cross(v);
    return a + k*u;
  }

 protected:
  PTR<Point> a, b, c, d;
 public:
  LineIntersection (Point *a, Point *b, Point *c,
                    Point *d)
    : a(a), b(b), c(c), d(d) {}
};
```

Convex Hull:Improved Textbook Algorithm I

```
#include "hull.h"

void convexHull (Points &p, Points &h)
{
  for (int i = 1; i < p.size(); ++i)
  if (XOrder(p[i], p[0]) == 1) {
    PTR<Point> t = p[i];
    p[i] = p[0];
    p[0] = t;
  }
  sort(p.begin() + 1, p.end(), CCWOrder(p[0]));
```

Convex Hull:Improved Textbook Algorithm II

```
    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;
      }
    }
}
```

Object Types

- An object type is a class whose elements are parameters.
- They can be stored as convenient.
- The class must define two methods:
  1. int size();
     returns the number of parameters.
  2. Parameter & operator[] (int i);
     returns a reference to the *i*th parameter.
- These methods are defined for Parameter, PV2, and PV3.

Circle Data

```
class CircleData {
 public:
  CircleData () {}
  CircleData (const PV2 &o, const Parameter &r)
    : o(o), r(r) {}
  int size () const { return 3; }
  Parameter & operator[](int i) {
    return i < 2 ? o[i] : r;
  }

  PV2 o;
  Parameter r;
};
```