Queens College, CUNY, Department of Computer Science

Numerical Methods

CSCI 361 / 761

Spring 2018

Instructor: Dr. Sateesh Mane

due Monday, Feb 12, 2018, 11.59 pm

1 Homework set 1

- **Experience has demonstrated that in many cases the source of difficulty is not the mathematics.**
- **The source of difficulty is the English (understanding the text).**
- **If you do not understand the words in the lectures or homework, THEN ASK.**
- **If you do not understand the concepts in the lectures or homework, THEN ASK.**
- **Send me an email, explain what you do not understand.**
- **Do not just keep quiet and then produce nonsense in exams.**

1.1 GCD

Euclid's algorithm to calculate the greatest common divisor (GCD) of two positive integers was presented in class. The code was given as follows

```
long gcd_euclid(long a, long b)
{
  if (a < b) {
    return gcd_euclid(b,a);
  }

  long c = (a%b);
  if (c == 0)
    return b;
  else if (c == 1)
    return 1;

  return gcd_euclid(b,c);
}
```

1.1.1

- **Modify the code to return 0 to guard against invalid inputs $a < 0$ or $b < 0$.**
- *Excel has a function "gcd(number1, number2)." Try it.*
  Observe that Excel will complain if you enter negative inputs.
  What if either a or b equal zero? If $a = 10$ and $b = 0$, then Excel returns $\gcd(10, 0) = 10$.
- Something to think about: if $a > 0$ and $b = 0$, then a obviously divides a, and logically, it also divides zero *(the remainder is zero!)*.
  Modify the above code so that if $a = 0$ then return $\gcd = b$, and if $b = 0$ then return $\gcd = a$. Obviously if $a = b = 0$ then $\gcd = 0$. You can check by comparing your function output to the Excel output, for arbitrary values of a and b.

1.1.2

The lowest common multiple (LCM) of two positive integers is related to the GCD via

$$LCM(a, b) \times GCD(a, b) = ab\,.$$

Write a function with signature "long LCM(long a, long b)" to compute the LCM of two integers a, b. If $a \leq 0$ or $b \leq 0$, then return 0.

1.2 Convert decimal to binary, decimal to hex

This function inputs a positive integer in decimal, and outputs an array (vector) with digits in any base from 2 through 16.

```
int dec_to_base(int a, int base, std::vector<char> & digits)
{
  char alphanum[16] = {'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};

  digits.clear();
  if ((base < 2) || (base > 16)) return 1;   // fail
  if (a < 0) return 1;                        // fail
  if (a == 0) {
    digits.push_back(alphanum[0]);
    return 0;
  }
  while (a > 0) {
    int rem = a % base;
    char c = alphanum[rem];
    digits.push_back(c);
    a /= base;
  }
  std::reverse(digits.begin(), digits.end());  // reverse the digits
  return 0;
}
```

- Note that the output vector is 'char' not 'int' because the digits are alphanumeric.
- The function return type is 'int' not 'void' because we test for bad inputs: the function returns 0 for success and 1 for failure.
- The output elements are ordered so that if we print as follows, we get the digits in the base

```
for (int i = 0; i < (int) digits.size(); ++i) {
  std::cout << digits[i];
}
std::cout << std::endl;
```

1.2.1

Try this out for sample inputs such as $a = 106$ (you should get 6A), etc.
Excel has a function "dec2hex." You can check by comparing the function output to the Excel output. If $a = -1$, Excel returns dec2hex(-1) = FFFFFFFFFF (on a 32-bit computer). Let us skip negative inputs. ***The lesson is that people who write "industrial strength" commercial products need to guard against all cases.***

1.3 Horner's rule (nested summation) and Taylor series

- The exponential function $\exp(x)$ can be expanded in a power series as

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

- It was explained in class how mathematical libraries compute $\exp(x)$.
- Here we shall write a simple function to compute $\exp(x)$ to an accuracy of 10^{-15} for $|x| \le 0.5$.
- First we truncate the above series to a finite sum up to $n_{\max}$ terms

$$S_{\exp} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^{n_{\max}}}{n_{\max}!} \,.$$

- This makes the sum a Taylor series (or a Maclaurin series), and also a polynomial in x.
- Using $n_{\max} = 30$ is sufficient for our purposes.
- **Write an efficient expression using nested sums to compute the above sum.**
- **Write a function to compute your sum.**
- The function signature is

```
double exp_sum(double x);
```

- Test your function using values of x where $-0.5 \le x \le 0.5$. (You will have to write a main program to call your function.) Compare your output to the library function value of $\exp(x)$. The absolute difference $|\texttt{exp\_sum}(x) - \exp(x)|$ should be always less than 10^{-15}.

1.4 Convex hull

Let us write a function to compute the convex hull of a set of n points in a plane.
Instead of carrying around too many arrays, let us employ the power of C++ and use a class.

- Write the following `Point` class. Make everything public for simplicity.

```
class Point {
public:
  Point(double x1, double y1) : x(x1), y(y1) {
    r = sqrt(x*x + y*y);
    if (r > 0.0) {
      theta = atan2(y,x);
    }
    else {
      theta = 0;
    }
  }

  double x;
  double y;
  double r;
  double theta;
};
```

- We compute the values of r and θ for later use in comparisons.
- This is the advantage of writing a class: we can store relevant information all in one object.
- As explained in the lectures, we shall require a comparison function. For points $\boldsymbol{a}$ and $\boldsymbol{b}$, we test if $\theta_a < \theta_b$. If $\theta_a = \theta_b$, we test if $r_a < r_b$.
- Write the following comparison function.

```
bool vcomp(const Point &a, const Point &b)
{
  if (a.theta < b.theta) {
    return true;
  }
  else if (a.theta > b.theta) {
    return false;
  }

  return (a.r < b.r);
}
```

Now we write the convex hull function. This is the function signature:

```
void convex_hull_func(int n, const std::vector<double> & x,
                             const std::vector<double> & y,
                             std::vector<Point> & convex_hull);
```

- The inputs are
  (i) int `n` (number of points)
  (ii) arrays `x` and `y` of type `double` (coordinates of points).
- The output is
  (iii) array `convex_hull` of type `Point`.
- **We need the input `n`.**
- Do not assume the length of the input arrays `x` and `y` is exactly `n`. ***They could be longer.***
- We could also make the input an array of `Point` objects.

The following pseudocode describes the steps. You can use it as the basis to write a working function.

1. Let us define a constant `tol` $= 10^{-14}$ for later use.
2. We begin by clearing `convex_hull` and testing for the trivial case $n \leq 3$.
   If $n \leq 3$ we populate `convex_hull` with the input coordinates and return.
   We insert the point $(x[0], y[0])$ at the end to close the polygon.

   ```
   const double tol = 1.0e-14;
   convex_hull.clear();
   if (n <= 0) return; // simple safeguard
   if (n <= 3) {
     for (int i = 0; i < n; ++i) {
       convex_hull.push_back( Point(x[i],y[i]) );
     }
     convex_hull.push_back( Point(x[0],y[0]) );  // close the polygon
     return;
   }
   ```

3. Next, find the minimum value $y_{\min}$ and save the partner value $x_{\min}$.
   Also store the index $i_{\min}$ where the minimum is located.
   Initialize the variables `int imin=0`, `double xmin=x[0]` and `double ymin=y[0]`.
   (a) Loop from $i = 1$ to $n - 1$.
   (b) Test if `ymin > y[i]`. If yes, then set `imin=i`, `xmin=x[i]` and `ymin=y[i]`.
   (c) Else test if `ymin == y[i]`. If yes, then test if `xmin > x[i]`. If yes, then set `imin=i`, `xmin=x[i]` and `ymin=y[i]`.

4. Now construct and sort the vectors $\boldsymbol{v}_i$. Do it as follows:

```
std::vector<Point> v;
for (int iv = 0; iv < n; ++iv) {
  double vx = x[iv] - xmin;
  double vy = y[iv] - ymin;
  if (iv == imin) {  // avoid roundoff error
    vx = 0;
    vy = 0;
  }
  v.push_back( Point(vx,vy) );
}

// sort the vectors
std::sort(v.begin(), v.end(), vcomp);
```

5. Note the following important details:

   (a) The special case `vx = 0` and `vy = 0` for `iv == imin` is to avoid roundoff error. This is the "starting point" of the convex hull.

   (b) In the lecture, there were only $n-1$ vectors $\boldsymbol{v}_i$.

   (c) Furthermore, the lecture also stated (for the starting point) "Without loss of generality, suppose this point is (x_1, y_1)."

   (d) This is impractical for computation. Instead, the code has n vectors $\boldsymbol{v}_i$, with $\boldsymbol{v}_i = (0,0)$ for `i = imin`. The comparison function `vcomp` has been formulated so that the sort algorithm will move that item to the first element in the sorted array.

6. Now construct the points $\boldsymbol{p}_i$. As stated in the lecture, $\boldsymbol{p}_i$ and $\boldsymbol{v}_i$ can share the same memory. Note that $\boldsymbol{p}_0 = (x_{\min}, y_{\min})$ automatically (from the sort).

```
// add (xmin, ymin) to sorted vectors
for (int ip = 0; ip < n; ++ip) {
  v[ip].x += xmin;
  v[ip].y += ymin;
}
std::vector<Point> &p = v;  // p is reference to v, shares same memory
```

7. Initialize the stack for the convex hull.

```
convex_hull.push_back(p[0]);
convex_hull.push_back(p[1]);
```

8. Next loop through the points $\boldsymbol{p}_i$ and update the stack. The tests are described in the lecture. However, they require vector calculus. It is unrealistic to ask you to code them on your own. I will give you the code for the loop of tests.
   We use the const parameter "`tol`" in the tests.

```
int i = convex_hull.size();
while (i < n) {
  // test for direction of rotation of edges
  int last = convex_hull.size() - 1;
  int second_last = last - 1;
  double ux = convex_hull[last].x - convex_hull[second_last].x;
  double uy = convex_hull[last].y - convex_hull[second_last].y;
  double wx = p[i].x - convex_hull[last].x;
  double wy = p[i].y - convex_hull[last].y;

  double cross_product = ux*wy - uy*wx;
  if (cross_product > tol) {
    // counterclockwise rotation = add to convex hull
    convex_hull.push_back(p[i]);
    ++i;
  }
  else if (fabs(cross_product) <= tol) {
    // straight line = replace old point by new point
    convex_hull.pop_back();
    convex_hull.push_back(p[i]);
    ++i;
  }
  else {
    // clockwise rotation = erase a point in the stack
    convex_hull.pop_back();
  }
}
```

9. After the loop is over, the stack contains the points of the convex hull.
   Close the convex hull polygon by adding $\boldsymbol{p}_0$ to the end.

```
convex_hull.push_back(p[0]);
```

10. We are done. Exit the function.

Write a main program to call the function. Generate some points and compute the convex hull. Plot the points (in a spreadsheet?). Plot the convex hull (closed polygon). See if it works.

- **The function will work even if some of the points are not distinct.**
- **The function will work even if ALL the points coincide.**

- If you know about C++ iterators, there is a more elegant way to write the code for the tests.
- We use a `const_reverse_iterator` to access the last and second last elements in the stack.
- We use "`const`" because we do not change the data inside the `Point` objects.
- **You should learn how to use C++ iterators.**

```
int i = convex_hull.size();
while (i < n) {
  // test for direction of rotation of edges
  std::vector<Point>::const_reverse_iterator cit = convex_hull.crbegin();
  const Point &last_pt = *cit;
  const Point &second_last_pt = *(++cit);
  double ux = last_pt.x - second_last_pt.x;
  double uy = last_pt.y - second_last_pt.y;
  double wx = p[i].x - last_pt.x;
  double wy = p[i].y - last_pt.y;

  double cross_product = ux*wy - uy*wx;
  if (cross_product > tol) {
    // counterclockwise rotation = add to convex hull
    convex_hull.push_back(p[i]);
    ++i;
  }
  else if (fabs(cross_product) <= tol) {
    // straight line = replace old point by new point
    convex_hull.pop_back();
    convex_hull.push_back(p[i]);
    ++i;
  }
  else {
    // clockwise rotation = erase a point in the stack
    convex_hull.pop_back();
  }
}
```

1.5 Convex hull #2

- I was impressed by the suggestion in class by a student, for a computationally cheaper sort function.
- The suggestion is equivalent to the use of the vector cross product.
- We do not need the angle θ, and we do not need to compute any square roots.
- Let us implement the sort using the vector cross product.
- To avoid confusion, let us define a new class `Point2`.
- You can rename the previous class `Point1` if you like.
- Write the following `Point2` class. Make everything public for simplicity.

```
class Point2 {
public:
  Point2(double x1, double y1) : x(x1), y(y1) {
    r2 = x*x + y*y;
  }

  double x;
  double y;
  double r2;
};
```

- The sort function is as follows. We are given input objects a and b.
  1. If $a_x b_y - a_y b_x > 0$, return `true`.
  2. If $a_x b_y - a_y b_x < 0$, return `false`.
  3. If $a_x b_y - a_y b_x = 0$, test if $a_x^2 + a_y^2 < b_x^2 + b_y^2$.
- Write the following comparison function.

```
bool vcomp2(const Point2 &a, const Point2 &b)
{
  double diff = a.x*b.y - b.x*a.y;
  if (diff > 0.0) {
    return true;
  }
  else if (diff < 0.0) {
    return false;
  }
  return (a.r2 < b.r2);
}
```

- **Everything else is the same as in Question 1.4.**
- Replace `Point` by `Point2` and `vcomp` by `vcomp2`.

- Here are some input points to test your code.
- I stated that the algorithm will work even if some of the points coincide.
- Let us have 10^4 points, each repeated 100 times, so a total of 10^6 points *(why not?)*.

```
std::vector<double> x_vec;
std::vector<double> y_vec;

int n = 10000;
for (int i = 0; i < 100*n; ++i) {
  x = cos(i%n);
  y = sin(3*(i%n)) * (1.0 - 0.5*x*x);

  x_vec.push_back(x);
  y_vec.push_back(y);
}
```

- On my computer, the original algorithm took 0.81 sec and the new algorithm took 0.67 sec.
- A plot of the points and the convex hull, using either sort function, is shown in Fig. 1.

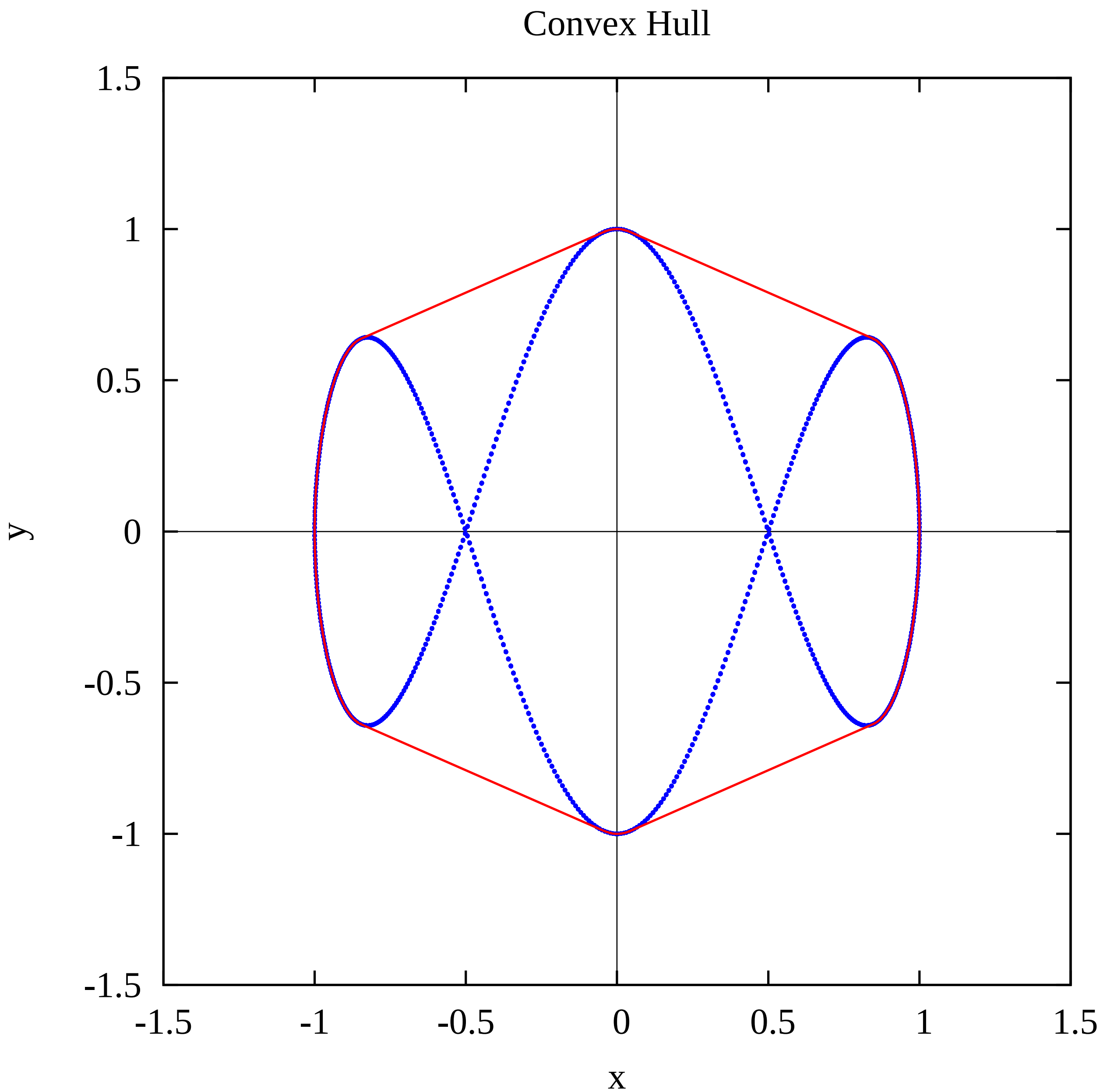

Figure 1: Plot of points in a plane and convex hull, using the algorithm in Question 1.4 or 1.5.