326.041 (2015S) – Practical Software Technology

(Praktische Softwaretechnologie)

Polynomials, Newton Approximation, Gradient Descent

Alexander Baumgartner
Alexander.Baumgartner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Polynomials

- A monomial (term) is a product of powers of variables with nonnegative integer exponents multiplied by a constant.
  - $-7x^7$ or $\frac{x^4}{3}$
  - $3x^3y^2z^5$ or $\frac{x_1^2x_2^3x_3}{\pi - 3\imath}$
- A polynomial is a sum of some monomials.
- **Univariate** polynomial: $-7x^7 + \frac{x^3}{2} - \frac{11x^2}{3} + 4x + 2$
- Implementation: How to represent a polynomial?
  - List of coefficients: $(2, 4, -\frac{11}{3}, \frac{1}{2}, 0, 0, 0, -7)$
  - Sorted map that maps exponents to coefficients: $\{7 \mapsto -7, 3 \mapsto \frac{1}{2}, 2 \mapsto -\frac{11}{3}, 1 \mapsto 4, 0 \mapsto 2\}$

Multivariate Polynomials

- **Multivariate** polynomial: $-7x^7y^2z^3 + \frac{x^3y^3z}{2} - \frac{11x^3}{3} + 4xz^9 + 2$
  - 3D matrix of coefficients is a bad idea to represent it.
- There are different kinds of monomial orders.
  - Sorted map that maps exponent vectors to coefficients:
    Lexicographic order:
    $(7,2,3) \mapsto -7,\quad (3,3,1) \mapsto \frac{1}{2},\quad (3,0,0) \mapsto -\frac{11}{3}$
    $(1,0,9) \mapsto 4,\quad (0,0,0) \mapsto 2$
  - Sorted map that maps **total degree** + exponent vector to coefficient:
    Graded lexicographic order:
    $(\mathbf{12},7,2,3) \mapsto -7,\quad (\mathbf{10},1,0,9) \mapsto 4,\quad (\mathbf{7},3,3,1) \mapsto \frac{1}{2},$
    $(\ \mathbf{3},3,0,0) \mapsto -\frac{11}{3},\quad (\ \mathbf{0},0,0,0) \mapsto 2$
  - Sorted map that maps **total degree** + exponent vector to coefficient:
    Graded reverse lexicographic order:
    - Compare total degree first.
    - Compare exponents of z and the monomial with the smaller one "wins"
    - Compare exponents of y and the monomial with the smaller one "wins"

    E.g., $-7x^7y^2z^3 + 4x^3y^9$ then $(12,3,9,0)$ comes before $(12,7,2,3)$
    Better computational behavior for Gröbner Basis computation.

Polynomial.java

- $SortedMap<int[], Double>\ \ monomials$

  Exponents ↗ (int[]) ↖ Coefficients (Double)

- Provide a **Comparator** to compare the exponent vectors. It defines a certain monomial order.

```
1  public class Polynomial {
2      private int numVars;
3      private SortedMap<int[], Double> monomials
4          = new TreeMap<>( new Comparator<int[]>() {
5              public int compare(int[] v, int[] w) {
6                  for (int i = 0; i < v.length; i++) {
7                      int cmp = w[i] - v[i];
8                      if (cmp != 0) return cmp;
9                  }
10                 return 0;
11             } });
12     public Polynomial(int numVars) {
13         this.numVars = numVars;
14     }
```

Add Monomial with Total Degree

- We use graded lexicographic order.
- The user does not need to provide a total degree (internal detail).

```
1  public void add(Double coeff, int... exponents) {
2      if (exponents.length != numVars)
3          throw new IllegalArgumentException("...");
4      int[] gradedExp = new int[numVars + 1];
5      int total = 0;
6      for (int i = 0; i < numVars;) {
7          int val = exponents[i];
8          total += val;
9          gradedExp[++i] = val;
10     }
11     gradedExp[0] = total;
12     addIntern(coeff, gradedExp);
13 }
```

- Shift the given exponent vector right.
- Compute the total degree and put it into position 0 of the vector.

Add Monomial / Polynomial

- If we can add a monomial then polynomial addition becomes trivial.

```
1  private void addIntern(Double coeff, int... exponents) {
2      Double oldCoeff = monomials.get(exponents);
3      if (oldCoeff != null)
4          coeff += oldCoeff;
5      if (coeff != 0)
6          monomials.put(exponents, coeff);
7      else if (oldCoeff != null)
8          monomials.remove(exponents);
9  }
10
11 public void add(Polynomial p2) {
12     if (other.numVars != numVars)
13         throw new IllegalArgumentException("...");
14     for (Entry<int[],Double> m : p2.monomials.entrySet())
15         addIntern(m.getValue(), m.getKey());
16 }
```

Using the class Polynomial.java

- Create a polynomial with two indeterminates:

```
Polynomial p1 = new Polynomial(2);
```

- Add some monomials:

```
p1.add( 1d, 2, 0); // x^2
p1.add( 2d, 1, 0); // x^2 + 2x
p1.add( 1d, 0, 2); // x^2 + 2x + y^2
p1.add( 1d, 0, 1); // x^2 + 2x + y^2 + y
p1.add( 1d, 0, 1); // x^2 + 2x + y^2 + 2y
p1.add(-3d, 0, 0); // x^2 + 2x + y^2 + 2y - 3
```

- Create another polynomial with two indeterminates:

```
Polynomial p2 = new Polynomial(2);
```

- Add some monomials...
- Add polynomial $p2$ to polynomial $p1$:

```
p1.add(p2);
```

Polynomial Multiplication

- Multiply each monomial with each another.
- Create a new polynomial which is the sum of all the monomial multiplications.

```
1  public Polynomial multiply(Polynomial p2) {
2    if (p2.numVars != numVars)
3      throw new IllegalArgumentException("...");
4    int expLen = numVars + 1;
5    Polynomial ret = new Polynomial(numVars);
6    for(Entry<int[],Double> m1 : monomials.entrySet()) {
7      for(Entry<int[],Double> m2 :p2.monomials.entrySet()){
8        int[] expNew = new int[expLen];
9        for (int i = 0; i < expLen; i++)
10         expNew[i] = m1.getKey()[i] + m2.getKey()[i];
11       ret.addIntern(m1.getValue()*m2.getValue(), expNew);
12     }
13   }
14   return ret;
15 }
```

Evaluate Polynomial

- Evaluate a multivariate polynomial at a given point.
  E.g.: $p(x, y, z) = -7x^7y^2z^3 + \frac{x^3y^3z}{2} - \frac{11x^2}{3} + 4xz^9 + 2$
  E.g.: $p(1, 2, 1) = -\frac{65}{3}$

```
1  public Double eval(Double... point) {
2      if (point.length != numVars)
3          throw new IllegalArgumentException("...");
4      Double ret = 0.0;
5      for (Entry<int[], Double> m : monomials.entrySet()) {
6          Double monomialEval = m.getValue();
7          for (int i = 0; i < numVars;) {
8              Double xi = point[i];
9              for (int exp=m.getKey()[++i]; exp > 0; exp--)
10                 monomialEval *= xi; // Compute power
11         }
12         ret += monomialEval;
13     }
14     return ret;
15 }
```

Computing Partial Derivative

- $\frac{\partial}{\partial x}\, x^2 + xy + 4y^3 = 2x + y \qquad \frac{\partial}{\partial y}\, x^2 + xy + 4y^3 = x + 12y^2$
- $SortedMap<int[], Double>\ \ monomials$

  Exponents ↗ ↖ Coefficients
- Create a method $d(int\ i)$ to compute the partial derivative $\frac{\partial}{\partial x_i}$ for a given polynomial p.
- Algorithm $d(int\ i)$:
  - For each monomial $(expArray, coef)$ do
    - $coef = coef * expArray[i]$
    - decrement $expArray[i]$ and the total degree $expArray[0]$

Computing Partial Derivative

- Computing the partial derivative of a polynomial is easy and only needs $O(n)$ time.

```
1 public Polynomial d(int i) {
2   Polynomial result = new Polynomial();
3   for (Entry<int[], Double> m : monomials.entrySet()) {
4     int[] exps = m.getKey().clone();
5     exps[i]--; exps[0]--;
6     result.addIntern(m.getKey()[i] * m.getValue(), exps);
7   }
8   return result;
9 }
```

Polynomial Equation – Motivation

- Approximate the positive real n th root of a positive real number $\sqrt[n]{x}$. E.g. $\sqrt{2}$ or $\sqrt[7]{111}$:
  - $\sqrt{2} = x$ or $\sqrt[7]{111} = x$
  - $2 = x^2$ or $111 = x^7$
  - $0 = x^2 - 2$ or $0 = x^7 - 111$
- Approximate the positive real root of the corresponding polynomial.
- Newton iteration converges very fast (quadratically).
  - It can be generalized to systems of multivariate polynomials.

Newton Iteration

- Given: A function f (e.g. a polynomial) and a starting point x_0.
- Apply the recursive function until a certain accuracy is reached:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- Example: Take $x^2 - 2$ and the starting point $x_0 = 2$.
  - From $(x^2 - 2)' = 2x$ follows $x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n}$.
  - $x_1 = x_0 - \frac{x_0^2 - 2}{2x_0} = 2 - \frac{2^2 - 2}{2 * 2} = 1.5$
  - $x_2 = 1.5 - \frac{1.5^2 - 2}{2 * 1.5} = 1.416\dot{6}$
  - $x_3 = 1.416\dot{6} - \frac{1.416\dot{6}^2 - 2}{2 * 1.416\dot{6}} \approx 1.414216$

Newton Iteration – Java

- Given: A univariate polynomial p and a starting point x_0.
- Apply newton iteration until a certain relative accuracy is reached.

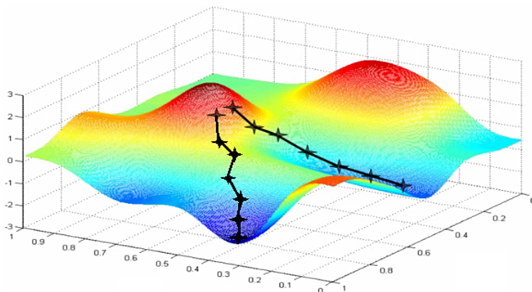
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

```
1 double newtonApprox(Polynomial p, double x_new) {
2     Polynomial dp = p.d(1);
3     double x_old;
4     do {
5         x_old = x_new;
6         x_new = x_old - p.eval(x_old) / dp.eval(x_old);
7     } while (Math.abs(x_old - x_new) > 10e-5);
8     return x_new;
9 }
```

- $Math.abs(x_old - x_new)$ is the termination condition.

Find Minimum

- Find (approximate) minimum of a function of multiple variables.
- Finding the minimum of a function f is the same as finding the maximum of $-f$.
- Idea: Start at some point $\vec{x}_0 = (x_1, \ldots, x_n)$ and go down the direction of the steepest slope step by step until the minimum is reached.



- Compute the gradient and follow the steepest direction.

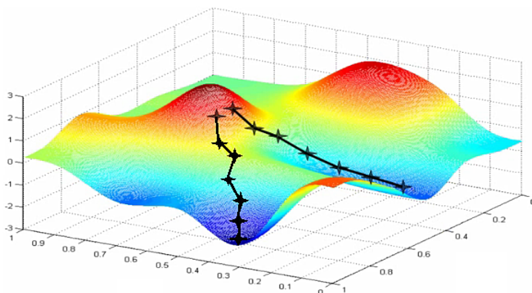
Computing the Gradient

- We compute the gradient ∇p from the partial derivatives of a multivariate polynomial p in the variables $x_1, \ldots, x_n$.

$$\nabla p = \begin{pmatrix} \frac{\partial p}{\partial x_1} \\ \cdots \\ \frac{\partial p}{\partial x_n} \end{pmatrix}$$

```
1 public Polynomial[] gradient() {
2     Polynomial[] grad = new Polynomial[numVars];
3     for (int i = 0; i < numVars; grad[i] = d(++i));
4     return grad;
5 }
```

Gradient Descent



- If $f(\vec{x})$ is differentiable in a neighborhood of $\vec{a}$, then it decreases fastest from $f(\vec{a})$ in the direction of the negative gradient $-\nabla f(\vec{a})$.
- For a given starting point $\vec{x}_0$ we apply the following recursive formula until the minimum is reached.

$$\vec{x}_{n+1} = \vec{x}_n - h_n \nabla f(\vec{x}_n)$$

- h_n is the step size.
- For the sake of simplicity we will use a fixed step size h.

Gradient Descent – Java

- Given: A multi polynomial p and a starting vector $\vec{x}_0$.
- Apply gradient descent until a certain accuracy is reached.

$$\vec{x}_{n+1} = \vec{x}_n - h_n \nabla f(\vec{x}_n)$$

```
1  Double[] gradDecent(Polynomial p, Double h, Double... x){
2      Polynomial[] grad = p.gradient();
3      Double sum;
4      do {
5          sum = 0.0;
6          for (int i = 0; i < grad.length; i++) {
7              Double val = grad[i].eval(x);
8              x[i] -= h * val;
9              sum += Math.abs(val);
10         }
11     } while (sum > 10e-5);
12     return x;
13 }
```

- sum is the steepness.

Polynomial Equation – Again

- Approximate the positive real nth root of a positive real number $\sqrt[n]{x}$. E.g. $\sqrt{2}$ or $\sqrt[7]{111}$:
  - $\sqrt{2} = x$ or $\sqrt[7]{111} = x$
  - $2 = x^2$ or $111 = x^7$
  - $0 = x^2 - 2$ or $0 = x^7 - 111$
  - $0^2 = (x^2 - 2)^2$ or $0^2 = (x^7 - 111)^2$
- Compute $\sqrt{2}$ by finding the minimum of $(x^2 - 2)^2$.
- Gradient descent can be used to approximate solutions of systems of multivariate polynomials.

Systems of Polynomial Equations

- Gradient descent can be used to **approximate solutions of a system of nonlinear equations**.

$$\begin{aligned} p_1(x_1,\dots,x_n) &= 0 \\ p_2(x_1,\dots,x_n) &= 0 \\ &\dots \\ p_n(x_1,\dots,x_n) &= 0 \end{aligned}$$

- The basic idea is to set $P = p_1^2 + p_2^2 + \cdots + p_n^2$ and apply gradient descent.

E.g.: $\left.\begin{aligned} x^2y - 2x &= 0 \\ y + 1 &= 0 \end{aligned}\right\}$ Solution $(x, y) \in \{(0, -1), (-2, -1)\}$

The polynomial $P = (x^2y - 2x)^2 + (y + 1)^2$ has a local minimum at the points $(0, -1)$ and $(-2, -1)$.

Using out Implementation

- Approximate the solutions of the simple example:

$$\left.\begin{array}{rcl} x^2y - 2x & = & 0 \\ y + 1 & = & 0 \end{array}\right\} \text{Solution } (x, y) \in \{(0, -1), (-2, -1)\}$$

- Create the polynomial $P = (x^2y - 2x)^2 + (y + 1)^2$:

```
1  Polynomial p1 = new Polynomial(2);
2  p1.add( 1d, 2, 1);          //  x^2 y
3  p1.add(-2d, 1, 0);          //  x^2 y - 2x
4  p1 = p1.multiply(p1);       // (x^2 y - 2x)^2
5  Polynomial p2 = new Polynomial(2);
6  p2.add( 1d, 0, 1);          //  y
7  p2.add( 1d, 0, 0);          //  y + 1
8  p2 = p2.multiply(p2);       // (y + 1)^2
9  p1.add(p2);                 // (x^2 y - 2x)^2 + (y + 1)^2
```

- Call the approximation:

```
1  result1 = gradDecent(p1, 0.01,  0.0,  0.0); //( 0,-1)
2  result2 = gradDecent(p1, 0.01, -1.0, -0.7); //(-2,-1)
```


Alexander.Baumgartner@risc.jku.at

Euclidean Division and GCD

- Common roots of given polynomials. $\iff$ Roots of the GCD.
- Euclidean division
  - **Input:** Two polynomials p_1 and $p_2 \neq 0$ in one variable x.
  - **Output:** The quotient polynomial q and the remainder polynomial r.
  - We denote by $\deg$ the total degree and by lc the coefficient of the leading monomial (leading coefficient).
    - $q = 0, r = p_1$
    - while $\deg(r) \geq \deg(p_2)$ do
    - $\quad s = \frac{\mathsf{lc}(r)}{\mathsf{lc}(p_2)} x^{\deg(r)-\deg(p_2)}$ // s is an auxiliary polynomial
    - $\quad q = q + s$
    - $\quad r = r - sp_2$
    - return (q, r)
- Greatest common divisor (GCD)
  - **Input:** Two polynomials p_1 and p_2 in one variable x.
  - **Output:** A polynomial that is a unique GCD of p_1 and p_2 up to multiplication by a constant.
    - Algorithm $\gcd(p_1, p_2)$:
    - $\quad$ if $p_2 = 0$ return p_1
    - $\quad$ else return $\gcd(p_2,$ remainder of $p_1/p_2)$

Exercise – Deadline June 19 th

- Generalize the class Polynomial.java from the lecture such that it works with coefficients from an arbitrary field.

```
1 public interface CoefficientField<T> {
2     T add(T val1, T val2);
3     T multiply(T val1, T val2);
4     T multiply(T val1, int n);
5     T invert(T val); // Multiplicative inverse
6     T negate(T val); // Additive inverse
7     T unitOne();
8     T unitZero();
9 }
```

- Implement one (approximate) representation of a field of your choice.
- Implement the euclidean division algorithm for univariate polynomials.
- Implement the euclidean algorithm to compute the GCD of two univariate polynomials over an arbitrary coefficient field.
- Make the monomial ordering adjustable.

See the guidance for this exercise on the Moodle page.