Case Studies: Netzberechnung (Hardy-Cross Verfahren). Numerische Integration.

OBJEKTORIENTIERTE PROGRAMMIERUNG ZWEI BEISPIELE

Live Coding

Einfache Beispiele zur Vererbung und Polymorphie.

Netzberechnung

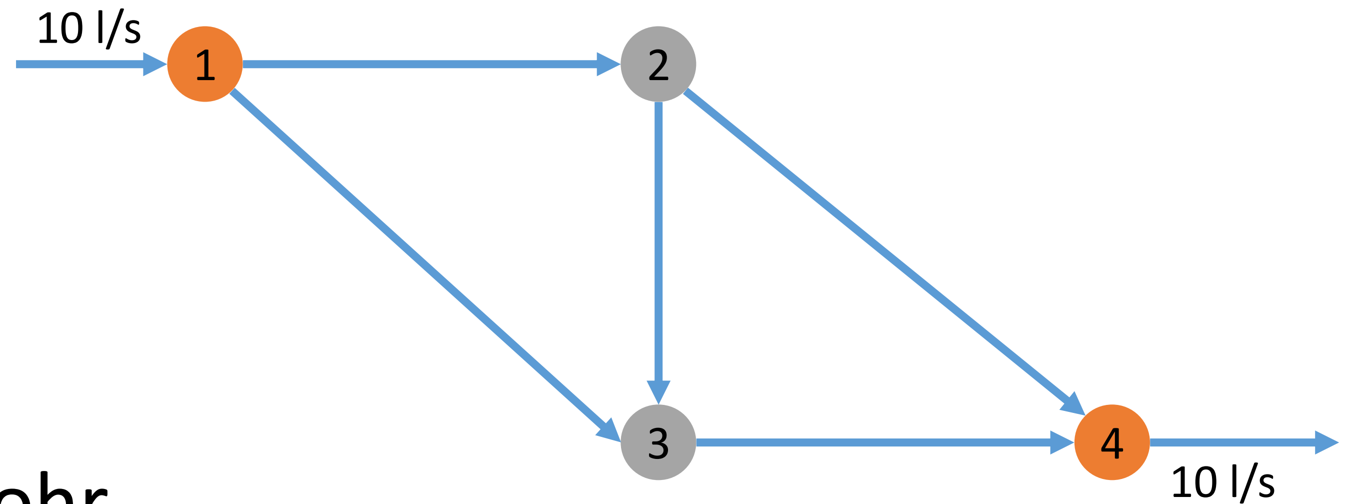
Bekannt:

Volumenstrom in und aus einem Rohrleitungsnetz

Rohrdicken, Längen und Rohrreibungszahlen

Benötigt:

Berechnung des Volumenstromes in jedem Rohr

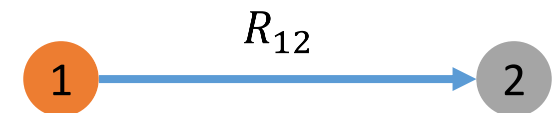


Druckverlustgleichung

Darcy-Weisbach Gleichung für die Verlusthöhe (Druckverlust) in einem Rohr p:

$$\mathrm{h_p(Q)} \coloneqq h_1 - h_2 = \underbrace{\frac{8 \cdot \lambda \cdot L}{g \cdot D^5 \cdot \pi^2}}_{r_p} \cdot Q^2$$

$\mathrm{h_p}$: Verlusthöhe
h_1, h_2: Ein- Ausgangsdruck
λ: Rohrreibungszahl
L: Rohrlänge
g: Erdbeschleunigung ≈9.8
L: Rohrdurchmesser
Q: Volumenstrom



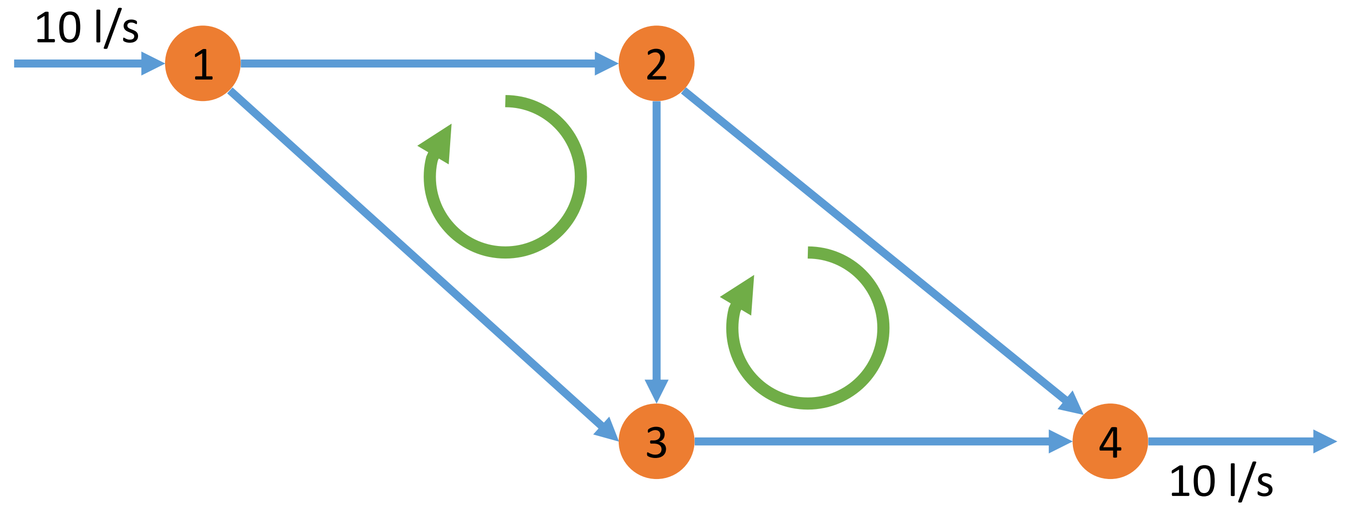
Invarianten

Energie-(Druck-)**erhaltung**

$$p_{12} + p_{23} - p_{13} = 0$$
$$p_{23} + p_{34} - p_{24} = 0$$

Massen-(Fluss-)**erhaltung**

$$Q_{12} + Q_{13} = Q_{in}$$
$$Q_{23} + Q_{13} = Q_{34}$$
$$Q_{12} = Q_{23} + Q_{24}$$
$$Q_{24} + Q_{34} = Q_{out}$$



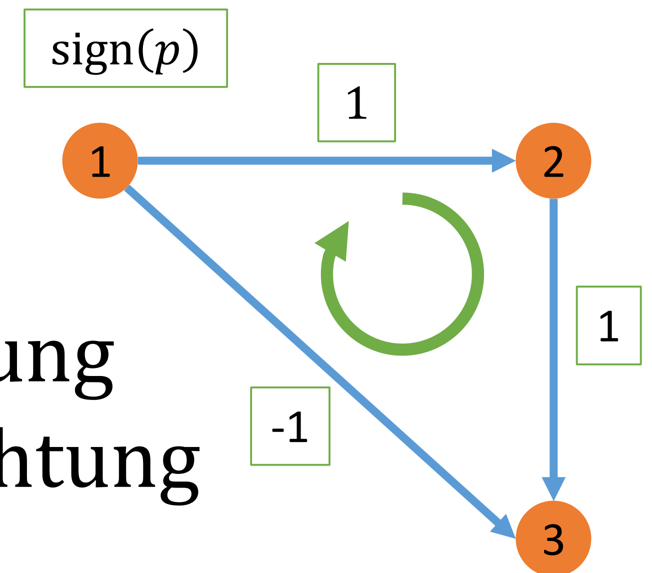
Energieerhaltung

Für Rohre p in einem Kreislauf K muss gelten:

$$\sum_{\mathrm{p} \in K} \operatorname{sign}(p) \cdot \mathrm{h_p}(Q_p) = 0$$

wobei

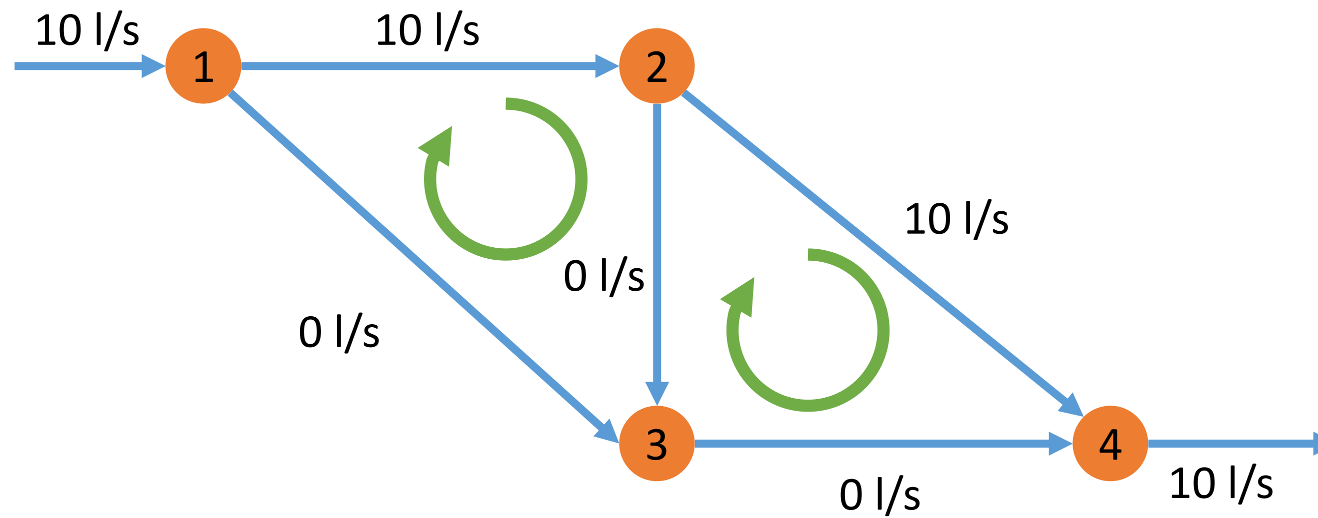
$$\operatorname{sign}(R) = \begin{cases} 1, & \text{wenn R in Uhrzeigerrichtung} \\ -1, & \text{wenn R entgegen Uhrzeigerrichtung} \end{cases}$$



Hardy Cross Verfahren

1. Rate einen flusserhaltenden Anfangszustand

z.B.:



Hardy Cross Verfahren

2. Für einen Kreislauf $\boldsymbol{K}$:

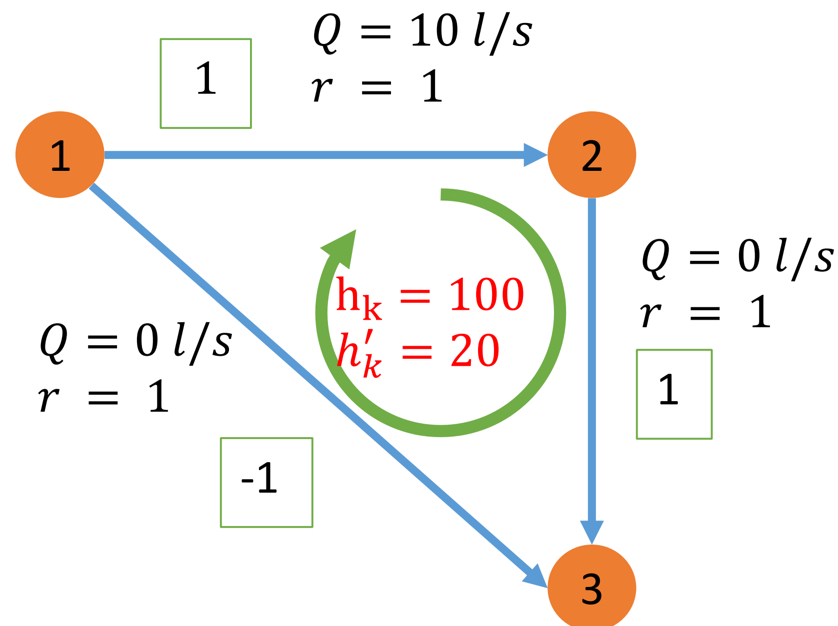
Berechne Druckverlust $\mathrm{h_k} = \sum_{\mathrm{p}\in K} sign(p) \cdot \mathrm{h_p}(Q_p)$

Berechne Ableitung $\mathrm{h}'_K = \sum_{p\in \mathrm{K}} h_p(Q_p)$

Dabei für Rohre:

$$\mathrm{h_p}(\mathrm{Q_p}) = r_p \cdot Q_p^2$$

$$\mathrm{h'_p}(\mathrm{Q_p}) = 2 \cdot r_p \cdot Q_p$$



Hardy Cross Verfahren

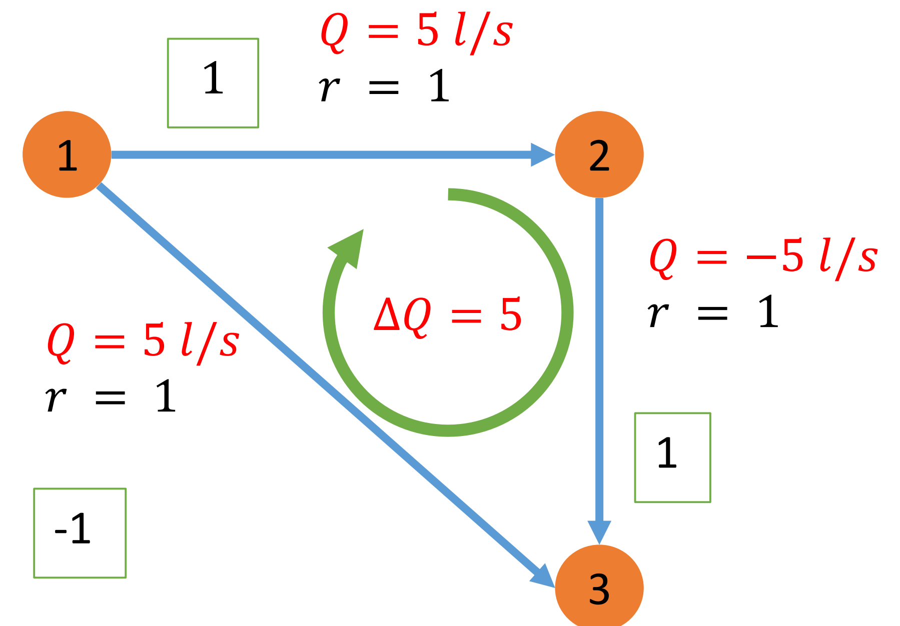
3. Eine Art Newton-Schritt

Berechne Flussänderung als

$$\Delta Q = \frac{\sum_{p \in K} sign(p) \cdot r_p \cdot Q_p^2}{\sum_{p \in K} 2 \cdot r_p \cdot Q_p}$$

Wende Flussänderung auf alle Pfade im Kreislauf K an:

$$Q_p \leftarrow Q_p - sign(p)\, \Delta Q$$



Hardy Cross Verfahren

Wiederhole 2 & 3 für alle Kreisläufe bis alle h_k **nahe genug bei 0.**

Das Verfahren funktioniert auch für Verallgemeinerte Rohre, wie z.B. Pumpen, Ventile, Turbinen etc.

Man benötigt lediglich jeweils die Funktionen

$\mathrm{h_p}(\mathrm{Q_p})$ und $\mathrm{h'_p}(\mathrm{Q_p})$.

Wie implementiert man das?

Modellierung

Einfache Modellierung

Wir gehen vereinfachend davon aus, dass

- Kreisläufe nicht berechnet, sondern vorgegeben werden
- Ein flusserhaltender Anfangszustand nicht berechnet, sondern vorgegeben wird
- Das Netz nur aus geschlossenen Kreisen besteht
- Die funktionalen Abhängigkeiten zwischen Volumenstrom und Druckverlust hinreichend gutartig sind und das Verfahren konvergiert.

Einfache Modellierung

Was benötigen wir?

- Ein Menge von Rohren (und ähnlichem).
  Wir nennen sie verallgemeinernd Kanten

- Eine Menge von Kreisläufen, bestehend aus Rohren.
  Rohre können in verschiedenen Kreisläufen mit jeweils anderer Richtung vorkommen. Ein Kreislauf enthält also eine Menge von Rohren zusammen mit den Richtungen.

Das wollen wir vermeiden:

```
private double[] X = new double[100];
private double[] Y = new double[100];
private int[] pipe = new int[100];
private int[][] inPipe = new int[10][100];
private int[][] outPipe = new int[10][100];
private int[] node = new int[100];
private int[] loop = new int[100];
private int[][] pipeNode = new int[100][2];
private int[][] flowDir = new int[100][10];
private int[][] pipeInLoop = new int[100][10];
private double[] roughness = new double[100];
private double[] diameter = new double[100];
private double[] flow = new double[100];
private double[] startFlow = new double[100];
private double[] flowChange = new double[100];
private double[] headLoss = new double[3];
private double[] netHeadLoss = new double[100];
private double flowCheck;
private double flowErr = 0.00001f;
```

```
for ( int p = 1; p < pCount+1; p++) {
if ( pipeInLoop[lp][pipe[p]] == lp) {
nrq = nrq + (double)(exp*roughness[pipe[p]]*Math.pow(flow[pipe[p]],exp-
1));
if (flowDir[lp][pipe[p]] == 1 ) {
rq = rq + (double)(roughness[pipe[p]]*Math.pow(flow[pipe[p]],exp));
} else {
rq = rq - (double)(roughness[pipe[p]]*Math.pow(flow[pipe[p]],exp));
}
```

Ein Beispiel von vielen: aus einem (ansonsten sehr nützlichen) Artikel* in einer wiss. Zeitschrift.

*Adeleke, Olawale, "Computer Analysis of Flow in the Pipe Network", Transnational Journal of Science and Technology February 2013 edition vol.3, No.2

Kanten müssen ...

```
public class Edge {
  public Edge(String Id, double initialFlow)

  public void SetFlow(double Q){...}
  public double Flow(){...}

  public double HeadLoss(double flow)
  public HeadLossDerivative(double flow)
}
```

... mit Initialwert für den Fluss versehen werden

... Ihren Fluss speichern!

... Druckverlust $h(Q)$ und Ableitung $h'(Q)$ in Abhängigkeit von Q angeben können

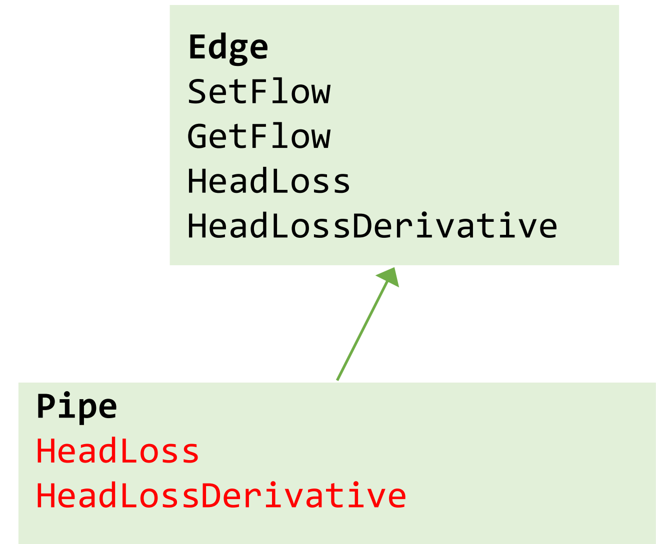
Rohre sind auch Kanten

```
public class Pipe extends Edge {
    double roughness;

    public Pipe(String id, double initialFlow, double length, double diameter, double friction) {
        super(id, initialFlow); // call constructor of Edge
        // Darcy-Weisbach:
        roughness = 8 * friction * length / 9.8 / Math.pow(diameter,5) / Math.PI / Math.PI;
    }

    public double HeadLoss(double flow) {
        return roughness * flow * flow;
    }

    public double HeadLossDerivative(double flow) {
        return 2*roughness * flow;
    }
}
```



Zugriff zum Fluss Q in der Superklasse Edge!

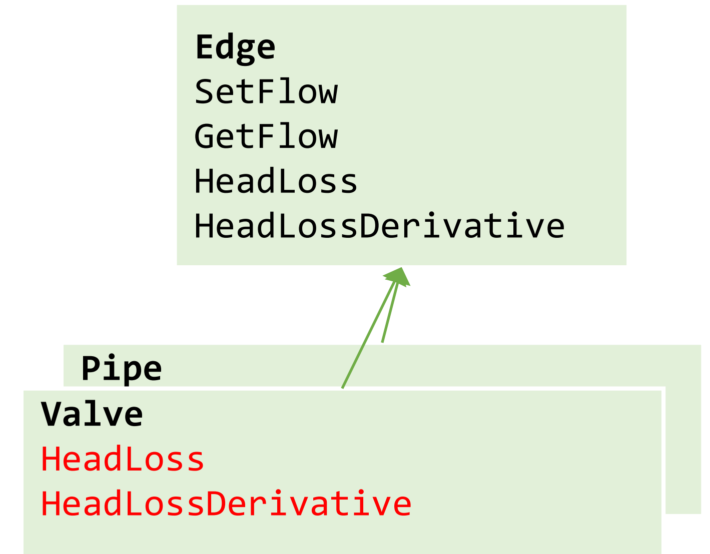
Ventile sind auch Kanten

```
public class Valve extends Edge {
   double headLoss;

   public Valve(String id, double initialFlow, double HeadLoss) {
      super(id, initialFlow); // call constructor of Edge
      headLoss = HeadLoss;
   }

   public double HeadLoss(double flow) {
      return headLoss;
   }

   public double HeadLossDerivative(double flow) {
      return 0;
   }
}
```



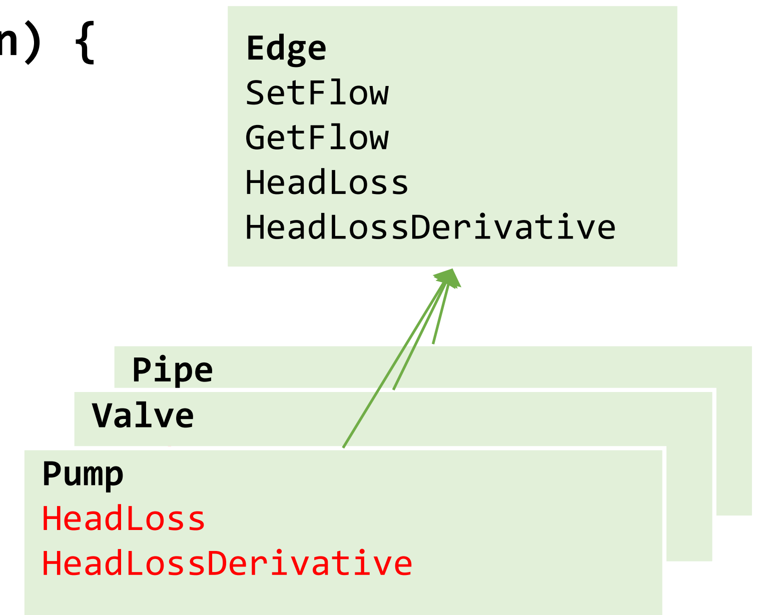
Pumpen sind auch Kanten

```
public class Pump extends Edge {
   double headGain;

   public Pump(String id, double initialFlow, double HeadGain) {
      super(id, initialFlow); // call constructor of Edge
      headGain = HeadGain;
   }

   public double HeadLoss(double flow) {
      return -headGain;
   }

   public double HeadLossDerivative(double flow) {
      return 0;
   }
}
```



Kreisläufe ...

```
import java.util.Vector;

public class Loop {
    public Vector<Edge> edges;
    public Vector<Integer> directions;

    public void Add(Edge edge, int direction){..}

    public double HeadLoss(){
        double x = 0;
        for (int i = 0; i < edges.size(); ++i) {
            Edge edge = edges.get(i);
            x += directions.get(i) * edge.HeadLoss(edge.Flow());
        }
        return x;
    }
    public double HeadLossDerivative(){..}
    public void ApplyDelta(double deltaFlow) {..}
}
```

.. müssen Kanten und jeweilige Flussrichtung speichern.

(java.util.Vector ist ein verallgemeinertes Array, welches wachsen kann)

... können schon hilfreiche Funktionen für den Hauptalgorithmus enthalten

Das allgemeine Setup...

```
public class Setup {
    public Vector<Loop> loops;
    public Loop AddLoop(Loop loop){...}

    public boolean compute(double accuracy){
        final int maxNumIterations = 100;
        boolean done = false;
        for (int iter = 0; iter < maxNumIterations && !done ; ++iter) {
            done = true;
            for (int i = 0; i<loops.size(); ++i) {
                Loop loop = loops.get(i);
                double delta  = loop.HeadLoss() / loop.HeadLossDerivative();
                loop.ApplyDelta(delta);
                if (Math.abs(loop.HeadLoss()) > accuracy)
                    done = false;
            }
        }
        return done;
    }
}
```

... besteht aus einer Menge von Kreisläufen

... und dem Hauptalgorithmus

Testen.

```
Setup setup = new Setup();
Edge p12 = new Pipe("12",10, 1.0, 0.1, 0.45);
...
Edge p34 = new Pump("34",0, 12.0);
Loop l1 = setup.AddLoop(new Loop());
l1.Add(p12,1); l1.Add(p23,1); l1.Add(p13,-1);
Loop l2 = setup.AddLoop(new Loop());
l2.Add(p24,1); l2.Add(p34, -1); l2.Add(p23, -1);

setup.print(); // Ausgabe aller Kanten
setup.compute(0.001); // Berechnung
setup.print(); // Erneute Ausgabe aller Kanten
```

Zusammengefasst

```
public abstract class Edge {
    public Edge( String Id, double initialFlow){
    public void SetFlow(double Q){
    public double Flow(){
    public abstract double HeadLoss(double flow);
    public double HeadLossDerivative(double flow)
    public void print()
}
```

enthält
"has a"

```
public class Loop {
    public Vector<Edge> edges;
    public Loop()
    public void Add(Edge edge, int direction)
    public double HeadLoss()
    public double HeadLossDerivative()
    public void ApplyDelta(double deltaFlow)
    public void print()
}
```

erbt von
"is a" relationship

```
public class Pipe extends Edge{
    public Pipe(String id, double initialFlow, double Length,
double Diameter, double FrictionFactor)
}
public class Pump extends Edge{
    public Pump(String id, double initialFlow, double HeadGain)
}
public class Valve extends Edge{
    public Valve(String id, double initialFlow, double HeadLoss)
}
```

enthält
"has a" relationship

```
public class Setup {
    public Vector<Loop> loops;
    public Setup()
    public Loop AddLoop(Loop loop)
    public boolean compute(double accuracy)
    public void print()
}
```

Fallstudie: Numerische Integration

Ziel: Erstellung eines Softwareframeworks zur numerischen Integration («Quadratur») einer gegebenen Funktion $f: \mathbb{R} \to \mathbb{R}$.

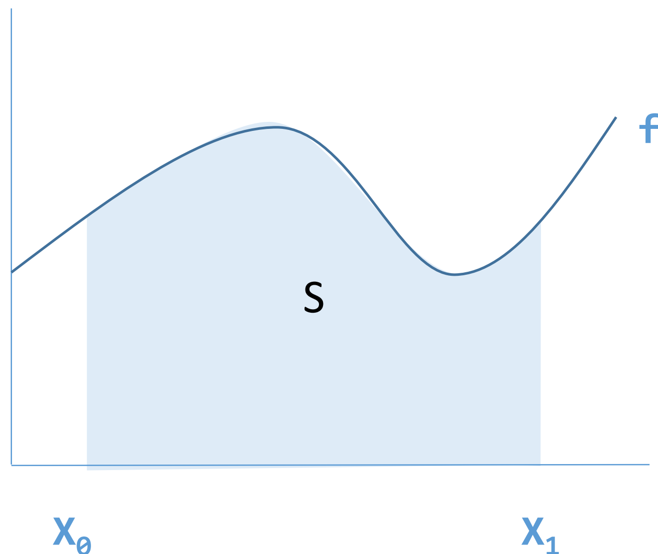
Problem aus Sicht der Softwareentwicklung: in Java gibt es keine Variablen von Funktionstyp. Wie können wir trotzdem ein generischen Tool bauen?

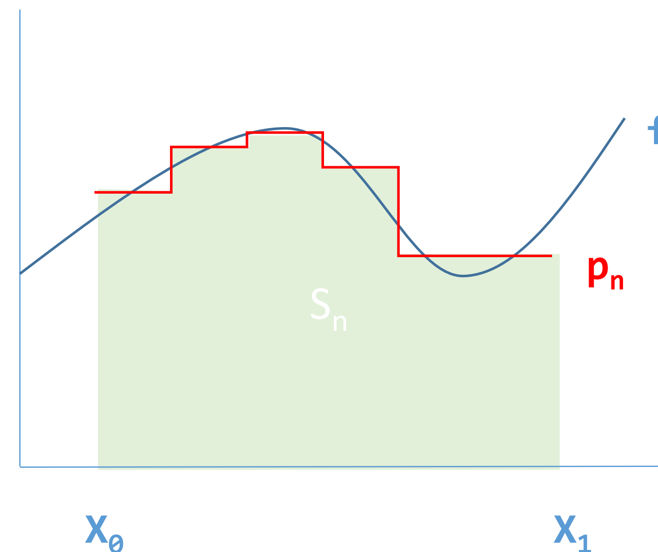
Antwort: wir verwenden Vererbung und Polymorphie für die generische Darstellung von reellwertigen Funktionen.

Numerische Integration

Einfachster Ansatz: Approximiere die Funktion f im gewünschten Integrationsintervall $[x_0, x_1]$ durch eine stückweise konstante Funktion p_n mit n Stücken.

Approximiere das Integral I von f durch das Integral I_n von p_n





Generische Funktionenklasse

```
public abstract class Function {
    public abstract double Evaluate(double x);
}
```

`abstract`:
Abstrakte Klassen können nicht instanziert werden
Abstrakte Funktionen benötigen keinen Funktionsrumpf, müssen jedoch von erbenden Klassen, welche nicht abstract sind, implementiert werden.

Generischer Integrierer

```
public abstract class Integrator {
   int n;
   public void SetNumberPieces(int pieces) {
      n = pieces;
   }
   public abstract double Integrate(Function f, double x0, double x1);
}
```

Konkretisierung: Parabel

$$f(x) = x^2$$

```
class Square extends Function{

    public double Evaluate (double x) {
        return x*x;
    }
}
```

Konkretisierung: Dichte der Normalverteilung

```
class Normal extends Function{
   double mu;
   double sigma;

   Normal(double m, double s)   {
      mu = m; sigma = s;
   }

   public double Evaluate(double x) {
      return  1/Math.sqrt(2*Math.PI)/sigma
              * Math.exp(-(x-mu)*(x-mu)/(2*sigma*sigma));
   }
}
```

$$\mathrm{f(x)} = \frac{1}{\sqrt{2\pi}\sigma}\,\mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Konkretisierung: Integrierer mit Rechteckregel

```
public class RectangleIntegrator extends Integrator {

    RectangleIntegrator(int pieces){
        n= pieces;
    }

    public double Integrate(Function f, double x0, double x1) {
        double sum = 0;
        double width=(x1-x0)/n;               // Intervallbreite
        for (int i = 0; i<n; ++i) {
            double x = x0 + (i+0.5)*width;    // Mitte des Intervalls
            double y = f.Evaluate(x);         // Abgreifen des Funktionswertes
            sum += y*width;                   // Rechteckinhalt addieren
        }
        return sum;
    }
}
```

Testbeispiel

```
Integrator integrator = new RectangleIntegrator(100);
Function sq = new Square();
Function N = new Normal(0,1);
System.out.println("I(sq,0,2)= " + integrator.Integrate(sq, 0, 2) );
for (int i=1; i<=3;++i){
      System.out.print("I(" +(-i) + "," + i + ")= ");
      System.out.println(integrator.Integrate(N,-i,i));
}
```

Ausgabe:

```
I(sq,0,2)= 2.6666000000000003
I(N,-1,1)= 0.6826975580161088
I(N,-2,2)= 0.954514133022569З
I(N,-3,3)= 0.9973041900876916
```

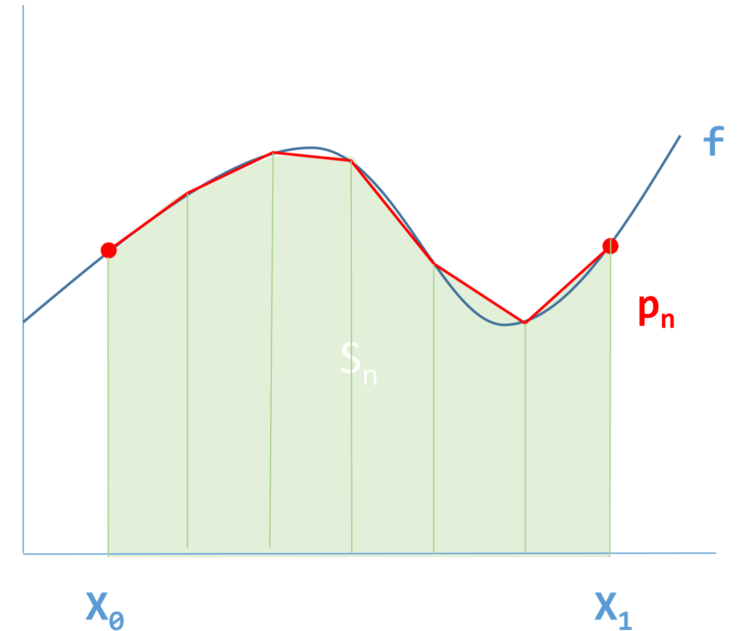
Deutung für normalverteilte Zufallsvariablen:

Innerhalb von 1 Standardabweichung liegen 68% der Daten

Innerhalb von 2 Standardabweichungen liegen 95% der Daten

Innerhalb von 3 Standardabwiechungen liegen 99.7% der Daten

Andere Integrationsregeln



Trapezregel: Stückweise affine Approximation

```
public class TrapezoidalIntegrator extends Integrator{
    TrapezoidalIntegrator(int pieces){
        n= pieces;
    }

    public double Integrate(Function f, double x0, double x1){
        double sum = 0;
        double width=(x1-x0)/n;
        for (int i = 0; i<n; ++i){
            double x = x0 + i*width;
            double y = f.Evaluate(x) + f.Evaluate(x+width);
            sum += y*width/2;
        }
        return sum;
    }
}
```

Beobachtung

Trapezregel ist für viele Funktionen kaum besser als die Rechteckregel.

Eine Inspektion des Algorithmus zeigt, dass Rechteckregel und Trapezregel auch nahezu dasselbe tun.

Für eine lineare Funktion stimmen die Regeln sogar überein.

Die Rechteckregel überschätzt in der Regel das Integral, die Trapezregel unterschätzt.

Beispiel: Numerische Integration von sin(x) im Intervall $[0, \pi]$

n	Rechteck	Trapez
1	3.14159	0
2	2.22144	1.57079
4	2.05234	1.89611
8	2.01290	1.97423
16	2.00321	1.99357
32	2.00080	1.99839
64	2.00020	1.99960
128	2.00005	1.99990

Präzisierung*

Betrachtung eines Einzelstückes der numerischen Integration am Intervall $[l, r]$.

Annahme: Funktion f genügend oft differenzierbar

Taylor-Entwicklung um $\tilde{x} = \frac{l+r}{2}$

$$f(x) = f(\tilde{x}) + (x - \tilde{x})f'(\tilde{x}) + \frac{(x - \tilde{x})^2}{2}\mathrm{f}''(\tilde{x}) + \frac{(x - \tilde{x})^3}{6}\mathrm{f}^{(3)}(\tilde{x}) + \frac{(x - \tilde{x})^4}{24}\mathrm{f}^{(4)}(\tilde{x}) + \ \ldots$$

Integration von f ergibt (mit $\Delta = r - l$)

$$\int_l^r f(x)dx = \Delta \cdot f(\tilde{x}) + \frac{\Delta^3}{24} \cdot f''(\tilde{x}) + \mathrm{O}(\Delta^5)$$

Präzisierung*

Seien $I(f)$ das exakte Integral, $I^R(f)$ und $I^T(f)$ die Approximationen mit Rechteck- / Trapezregel.

Dann gelten

$$I^T(f) = I(f) - \frac{\Delta^3}{24} f''(\tilde{x}) + O(\Delta^5)$$

$$I^R(f) = I(f) + \frac{\Delta^3}{12} f''(\tilde{x}) + O(\Delta^5)$$

Der Fehler ist immerhin mit der dritten Potenz der Länge der Stücke kontrollierbar

Das verhilft zu folgendem **Trick**

$$2 \cdot I^R(f) + I^T(f) = 3 \cdot I(f) + O(\Delta^5)$$

Noch besser: Der Fehler fällt mit der 5ten Poztenz!!

Daraus folgt die **Simpson Regel**

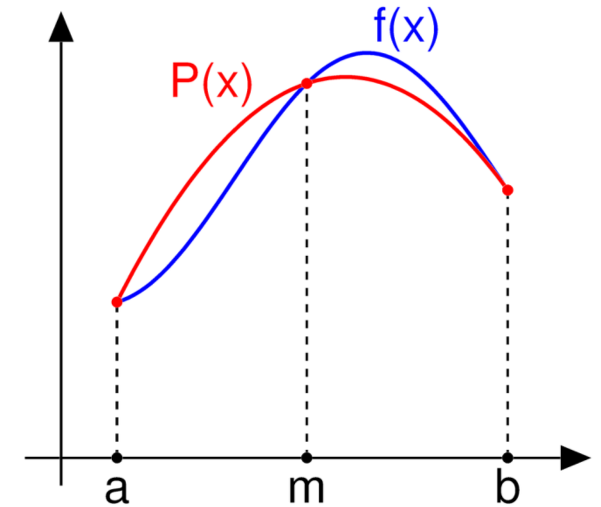
$$I(f) \approx I^S(f) := \frac{r-l}{6}\left(f(x_l) + 4f\left(\frac{r+l}{2}\right) + f(x_r)\right)$$

Simpson Regel*

Die Simpson-Integration korrespondiert mit quadratischer Interpolation von f

```
public class SimpsonIntegrator extends Integrator{
    SimpsonIntegrator(int pieces){
        n= pieces;
    }

    public double Integrate(Function f, double x0, double x1) {
        double sum = 0;
        double width=(x1-x0)/n;
        for (int i = 0; i<n; ++i)
        {
            double x = x0 + i*width;
            double y = f.Evaluate(x) +f.Evaluate(x+width) + 4*f.Evaluate(x+width/2);
            sum += y*width/6;
        }
        return sum;
    }
}
```



Quadratische Interpolation

Quelle: Wikipedia

Newton Cotes Formeln*

Die Newton-Cotes Formeln sind numerische Quadraturformeln zur approximativen Berechnung von Integralen durch Interpolation von Funktionen mit Lagrange Polynomen.

Rechteck-, Trapez- und Simpson-Regeln sind Beispiele dieser Formeln.

Eine andere berühmte numerische Integrationsmethode ist die Gauss-Quadratur

Darüber hinaus sind Verfahren interessant, die das Abtastgitter in x-Richtung adaptiv wählen. Solche Verfahren sind wichtig bei Funktionen, die nicht überall gleich «glatt» sind.

Monte Carlo Integration

Nach dem Gesetz der grossen Zahlen konvergiert das empirische Mittel von identischen, unabhängig gezogenen Zufallsvariablen gegen den Erwartungswert der Zufallsvariablen*.

Es gilt also

$$f_n = \frac{1}{n}\sum_{i=1}^{n} f(X_i) \rightarrow_{f.s.} \mathbb{E}(f(X)) = \int_{-\infty}^{\infty} f(x)d\mu(x)$$

Wenn man die Zufallszahlen X_i geichverteilt im Intervall $[x_0, x_1]$ zieht, so kann man das Integral von f also wie folgt approximieren

$$\int_{x_0}^{x_1} f(x)dx \approx (x_1 - x_0) \cdot \frac{1}{n}\sum_{i=1}^{n} f(X_i)$$

*in Wahrscheinlichkeit, sogar fast sicher

Monte Carlo Integration

Berechnet Mittelwert der Funktionswerte über dem Intervall *durch Sampling* und gibt dessen Produkt mit der Intervallbreite zurück.

```
public class MonteCarloIntegrator extends Integrator{

    MonteCarloIntegrator(int pieces){
        n= pieces;
    }

    public double Integrate(Function f, double x0, double x1) {
        double sum = 0;
        for (int i = 0; i<n; ++i) {
            double x = x0 + Math.random()*(x1-x0);
            sum += f.Evaluate(x);
        }
        return (x1-x0)*sum/n;
    }
}
```

Experiment: Vergleich der Verfahren

Approximation des Integrals $\int_0^{\pi} \sin(x)\,dx = 2$

n	Rechteck	Trapez	Simpson	MonteCarlo
1	3.14159265	0	2.0943951	1.87890144
2	2.22144147	1.57079633	2.00455975	1.51525667
4	2.05234431	1.8961189	2.00026917	2.0769775
8	2.01290909	1.9742316	2.00001659	1.8314429
16	2.00321638	1.99357034	2.00000103	1.76335613
32	2.00080342	1.99839336	2.00000006	1.96607781
64	2.00020081	1.99959839	2	2.06425794
128	2.0000502	1.9998996	2	1.94378659
256	2.00001255	1.9999749	2	2.01618607
512	2.00000314	1.99999373	2	2.03129742
1024	2.00000078	1.99999843	2	1.9790763

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Kapselung, Information Hiding

Verbergen des Zustands und der Implementierungsdetails von Objekten

Definition einer Schnittstelle zum Zugriff auf interne Datenstruktur
→ Abstraktion

Ermöglicht das Sicherstellen von Invarianten

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Vererbung

Objekte können Eigenschaften von Objekten erben

Abgeleitete Objekte können neue Eigenschaften besitzen oder vorhandene überschreiben

Macht Code- und Datenwiederverwendung möglich

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Polymorphie

Ein Bezeichner kann abhängig von seiner Verwendung unterschiedliche Datentypen annehmen.

Unterschiedliche Datentypen können bei gleichem Zugriff auf ihr gemeinsames Interface verschieden reagieren.

Macht „nicht invasive” Erweiterung von Bibliotheken möglich.