Kurs OMSI
im WiSe 2010/11

Objektorientierte Simulation mit ODEMx

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage

fischer|ahrens|eveslage@informatik.hu-berlin.de

4. ODEMx-Modul Synchronisation:
Bin, Res

- Verwaltung geteilt genutzter Ressourcen mittels Bin und Res
- Die Klasse Bin
- Behandlung von Unterbrechungen
- Die Klasse Res
- Beispiel: Autofährbetrieb (Einsatz von Bin und Res)

Unterbrechung

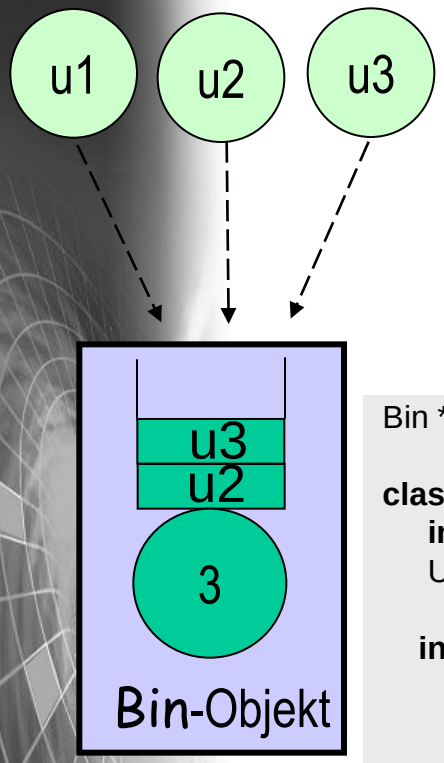
…

- des Wartevorgangs auf verfügbare Ressourcen (Token)
- der Benutzung der entnommenen Ressourcen (Token)

In beiden Fällen mittels interrupt !

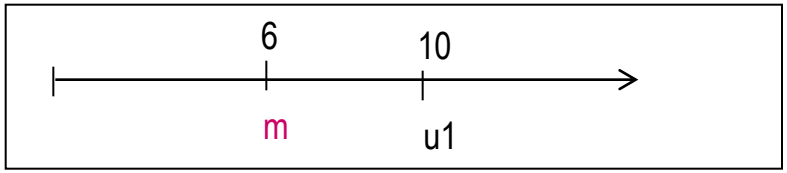
```
int res= service-> take(…); // liefert 0 Token, falls
                            // Blockierung unterbrochen worden ist
                            // sonst Anzahl der entnommenen Token
```

Variante B: Unterbrechung des Wartevorgangs



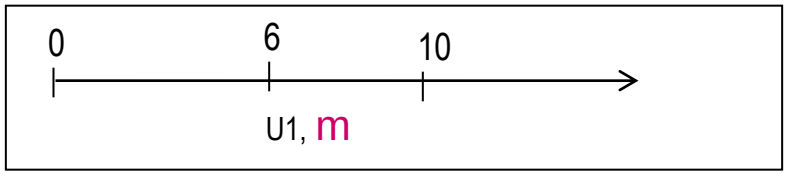
User-Objekte,
die gleichzeitig
gestartet werden

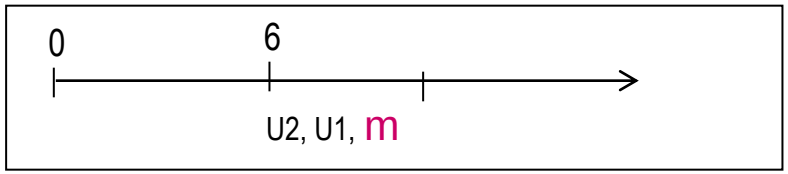
ein weiterer Prozess **m**
zum Zeitpunkt 6.0




```
class Manager:
        public Process {
   ...

   int Manager::main() {
     ...
       u1->interrupt();
     ...
     }
}
```

```
Bin *service= new Bin ("service", 3);

class User: public Process {
    int no;
    User (int n), no(n), Process (…) { ...};

   int User::main() {
      int ret;

      ...
      ret= service-> take(no);
      if (ret) {
         holdFor (10.0);
         service-> give(2);
      } else  ….
                  evtl. Nutzung eines anderen
                  Bin-Objektes nach Warteabbruch
    }
}
```





4. ODEMx-Modul Synchronisation:
Bin, Res

- Verwaltung geteilt genutzter Ressourcen mittels Bin und Res
- Die Klasse Bin
- Behandlung von Unterbrechungen
- Die Klasse Res
- Beispiel: Autofährbetrieb (Einsatz von Bin und Res)

Besonderheiten von Res

- Konstruktor
  sowohl die **initiale**(>=0) als auch **maximale** Tokenzahl (>0) ist einzustellen

- **unsigned int** acquire(n), n>0
  - unterbrechbare Warteaktion (evtl. mit Null-Zeit als „Durchläufer“)
  - ohne Unterbrechung wartet der Rufer solange, bis n Token verfügbar sind
  - Rückgabewert zeigt Unterbrechung an: 0, sonst n

- **unsigned int** release(n), n>0
  - Eingabe von Token
  - Token müssen nicht unbedingt diesem Res–Objekt vorab entnommen sein
  - wird maximale Tokenanzahl überschritten:
    Fehlermitteilung und Fortsetzung bei Ignorierung der überschüssigen Token

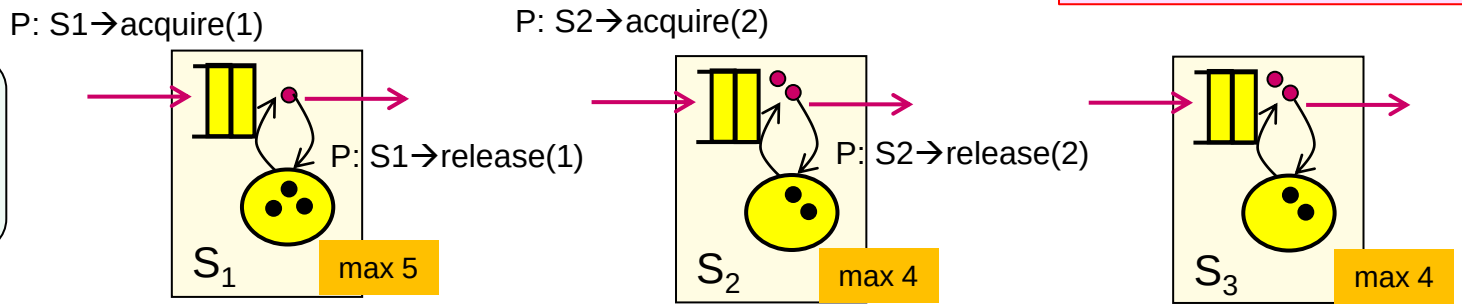
- **Sonderfunktionen zur dynamischen Korrektur der maximalen Token-Zahl**
  - **void** control(n), n>0 Erhöhung der maximalen Token-Anzahl
    bei Aktivierung des nächsten wartenden Prozesses

  - **void** uncontrol(n), n>0 Reduktion
    falls n Token überhaupt noch verfügbar sind, sonst Anpassung von n

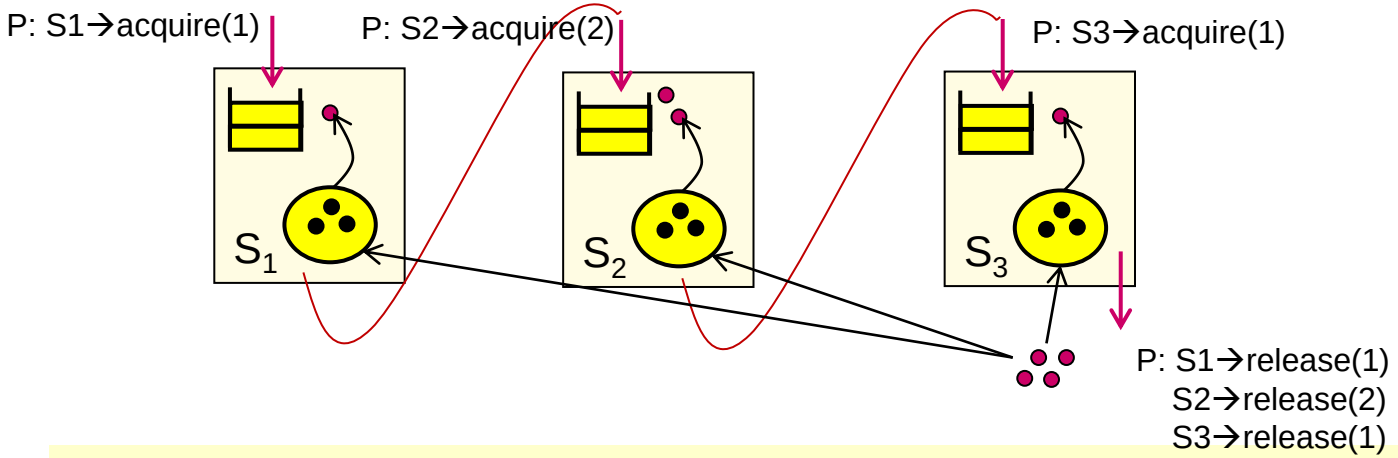
Res-Anwendungen

Token haben jedoch weder Identität noch Typ

- freie Ressource
- belegte -“-



Fall a) Prozesse bewegen sich durch Stationsketten (Res-Objekte), benötigen/benutzen Ressourcen einer Ressource und geben diese nach der Nutzung an die aktuelle Station komplett zurück



Fall b) Prozesse benötigen Ressourcen unterschiedlicher Sorten (Res-Objekte) für einen Arbeitsgang

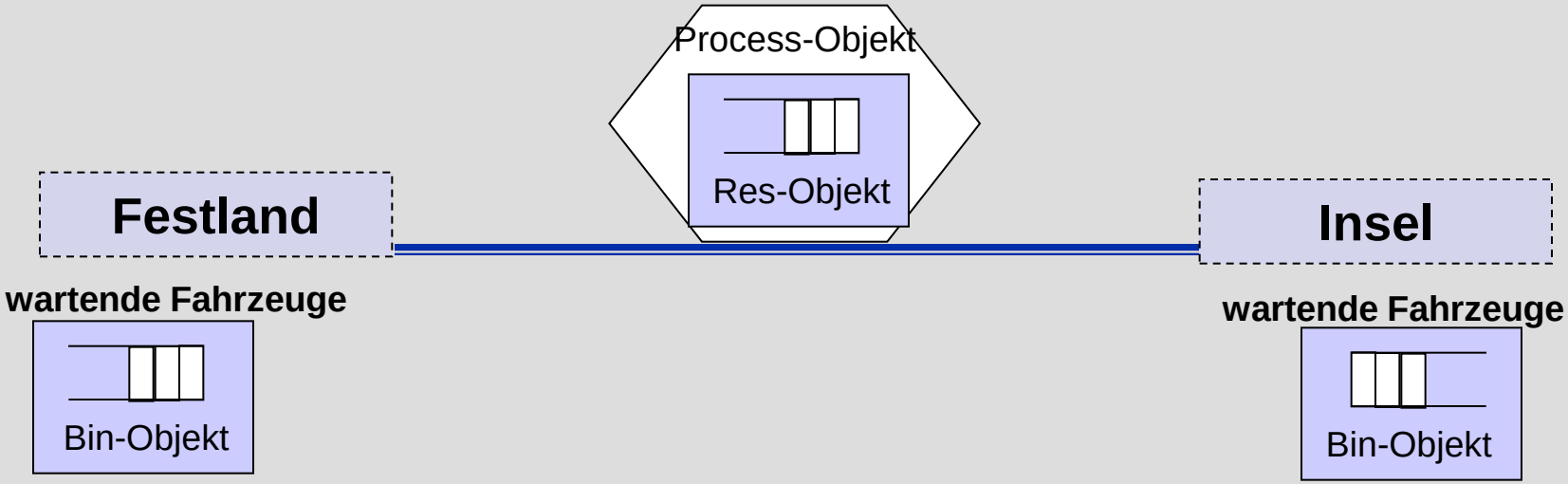
4. ODEMx-Modul Synchronisation:
Bin, Res

- Verwaltung geteilt genutzter Ressourcen mittels Bin und Res
- Die Klasse Bin
- Behandlung von Unterbrechungen
- Die Klasse Res
- Beispiel: Autofährbetrieb (Einsatz von Bin und Res)

Beispiel: Autofähre

- Erfassung von Bewertungsgrößen

1. Auslastung der Fähre im Report-File (Res-Objekt)
2. Beladungsprofil (Tally-Objekt)



3. separate Zählung der Leer- und Frachtfahrten (Count-Objekte)
4. Warteschlangenprofil für beide Warteschlangen (Bin-Objekte ins Report-File)

Beispiel: Autofähre

```
/*** Beispiel : Auto-Faehre ***/

#include <odemx/odemx.h>
using namespace odemx;

Simulation* sim =
            getDefaultSimulation();


//globale Variablen, nicht initialisiert
Bin    *q [2];
Res    *fload;

Rdist *next [2];
Rdist *crossing;

Tally *load;
Count *trips, *empties;

double minStayTime;


//Anwendungsklassen
class Ferry : public Process {
...
};

class Arrival : public Process {
...
};
```

```
int main ( unsigned int argc, const char* argv[]) {
      //Report-File-Festlegung
            HtmlReport myReport(sim, "Ferry_Report-2.html");

      //(dynamische)Trace-Konfiguration
            HtmlTrace trace (sim, "Ferry_Trace.html");
            trace.setFilter("all; mtn:changeState, changeExTime, \
                        execute process, init , \
                        changeTokenNumber  ");
            sim->addConsumer(&trace);
            sim->startTrace();

      //Zufallszahlengeneratoren, Initialisierung
            next[0]      = new Negexp(sim, "mainland", 0.1);
            next[1]      = new Negexp(sim, "island",   0.1);
            crossing     = new Normal(sim, "crossing", 7.5, 0.5);
      //Ressourcenverwaltung, Initialisierung
            q[0]         = new Bin(sim, "mainland",3);
            q[1]         = new Bin(sim, "island",1);
            fload        = new Res(sim, "ferry load", 4, 4);
      //Modellbeobachtung, Initialisierung
            trips        = new Count(sim, "trips");
            empties      = new Count(sim, "empty trips");
            load         = new Tally(sim, "av.load");

            minStayTime= 5.0;

            ...
```

Beispiel: Autofähre

```
/*** Beispiel : Auto-Faehre***/

#include <odemx/odemx.h>
using namespace odemx;

Simulation* sim =
            getDefaultSimulation();


//globale Variablen (nicht initialisiert
Bin    *q [2];
Res    *fload;

Rdist *next [2];
Rdist *crossing;

Tally *av_load;
Count *trips, *empties;

double minStayTime;


//Anwendungsklassen
class Ferry : public Process {
...
};

class Arrival : public Process {
...
};
```

```
      // (dynamische) Report-File-Konfiguration
            myReport.addProducer(next[0]);
            myReport.addProducer(next[1]);
            myReport.addProducer(q[0]);
            myReport.addProducer(q[1]);
            myReport.addProducer(fload);
            myReport.addProducer(crossing);
            myReport.addProducer(trips);
            myReport.addProducer(empties);
            myReport.addProducer(av_load);
       // Erzeugung von Process-Objekten und
       // Initialisierung des Prozessterminkalenders
            Arrival *a1 = new Arrival(0);  a1->activate();
            Arrival *a2 = new Arrival(1);  a2->activate();
            Ferry *f = new Ferry (fload);
            f->activateAt(7*60); //Start um 7.00 Uhr
       // Start der Simulation um 00:00 Uhr
            sim->runUntil(7*60 + 20);
       // Rückkehr nach 20min Fährbetrieb
      // um die Trace-Aufzeichnung abzuschalten
            sim->pauseTrace();
      // Fortsetzung des Fährbetriebes

            sim->run();
      // Abbruch der Simulation
      // wird von der Fähre erwartet
            myReport.generateReport();
            return (0);
}
```

Beispiel: Autofähre

```
/*** Beispiel : Auto-Faehre***/

#include <odemx/odemx.h>
using namespace odemx;

Simulation* sim =
            getDefaultSimulation();


//globale Variablen (nicht initialisiert
Bin    *q [2];
Res    *fload;

Rdist *next [2];
Rdist *crossing;

Tally *av_load;
Count *trips, *empties;

double minStayTime;


//Anwendungsklassen
class Ferry : public Process {
...
};

class Arrival : public Process {
...
};
```

```
class Arrival : public Process {
public:
    int  side; //index für q und next
   Arrival (int s): Process(sim,"Arrival"), side(s) {}
   int main ();
};
```

```
int Arrival::main() {
      for (;;) {
            holdFor (next[side]->sample());
            ::q[side]->give(1);
      }
      return 0;
}
```

Beispiel: Autofä

```
/*** Beispiel : Auto-Faehre***/

class Ferry : public Discrete {
            Res* myLoad;
public:
     int main ();
     Ferry(Res* r) : Process(sim,"Ferry")
            myLoad(r) {}
};

Rdist *next [2];
Rdist *crossing;

Tally *av_load;
Count *trips, *emp

double minStayTin

//Anwendungsklassen
class Ferry : public Process {
...
};

class Arrival : public Process {
...
};
```

Problem:
in dieser Zeitspanne eintreffende Autos kommen nicht mehr auf die Fähre

```
int Ferry::main() {
    do {
       for (int side=0; side<=1; side++) {
            double entryTime= sim->getTime();
// entladen
            int currentLoad = myLoad->getTokenLimit() -
                                    myLoad->getTokenNumber();
            holdFor (currentLoad*0.5);
            myLoad->release(currentLoad);
// beladen
            while ( myLoad->getTokenNumber()>0 &&
                        ::q[side]->getTokenNumber()>0 ) {
                  ::q[side]->take(1);
                  holdFor(0.5);
                  myLoad->acquire(1);
            };
// Mindestaufenthalt
            double loadTime = sim->getTime() - entryTime;
            if (loadTime < minStayTime)
                        holdFor(minStayTime - loadTime);
// Beladungsstatistiken
            currentLoad = myLoad->getTokenLimit() -
                                    myload->getTokenNumber();
            av_load->update(currentLoad);
            if (currentLoad == 0) empties->update(1);
// ueberqueren
            hold(crossing->sample());
       }
       trips->update(1);
    }
    while (sim->getTime() < 21*60 + 45); // nur tagsüber
    . . .
```

Beispiel: Autofähre

```
/*** Beispiel : Auto-Faehre***/

class Ferry : public Discrete {
            Res* myLoad;
public:
     int main ();
     Ferry(Res* r) : Process(sim,"Ferry")
            myLoad(r) {}
};

Rdist *next [2];
Rdist *crossing;

Tally *av_load;
Count *trips, *empties;

double minStayTime;


//Anwendungsklassen
class Ferry : public Process {
...
};

class Arrival : public Process {
...
};
```

```
    // (dynamische) Report-File-Konfiguration
            report.addProducer(next[0]);
            report.addProducer(next[1]);
            report.addProducer(q[0]);
            report.addProducer(q[1]);
            report.addProducer(fload);
```

```
. . .
   // zum Schluss noch einmal entladen
        int currentLoad = myLoad->getTokenLimit() -
                                  myLoad->getTokenNumber();
        holdFor(currentLoad*0.5);
        myLoad->release(currentLoad);

   // Abbruch der Simulation, zurück ins Hauptprogramm
   // Abbruchzeitpunkt=  21.45 Uhr + x

        sim->exitSimulation();
        return 0;
}
```

```
            sim->run();
    // Abbruch der Simulation
    // wird von der Fähre erwartet
            report.generateReport();
            return (0);
}
```

5. ODEMx-Modul Random

1. Charakterisierung von Zufallsgrößen
2. Approximation von Zufallszahlen
3. ODEMx- Zufallszahlengeneratoren (Übersicht)
4. Einstellung von Startwerten
5. Protokollierung
6. Berechnung von Zufallszahlen ausgewählter Verteilungen

Rolle von Wahrscheinlichkeit und Statistik *in der Simulation*

- Einflüsse der Systemumgebung oder Systemabläufe selbst unterliegen häufig dem Zufall

- Simulationsergebnisse sind dann als Stichprobe eines statistischen Experiments anzusehen

- Auf der Grundlage vieler Stichproben (Stichprobenraum) können statistische Kennwertprofile und Konfidenzaussagen (Aussagen zur Zuverlässigkeit der Ergebnisse) abgeleitet werden

- **wichtig**: der Einfluss des Zufalls muss im Simulationsexperiment wiederholbar dargestellt werden können
  (Test von Simulationsmodellen)

Zufallszahlen im Original und Modell

1. Realität ←→ Modell

reale Zufallsgrößen werden durch mathematische Modelle (Funktionen) approximiert:

- *Verteilungsfunktion, Dichtefunktion, statistische Kenngrößen*

2. Modell ← → Modell

es bestehen mathematische Zusammenhänge zwischen einzelnen Verteilungsfunktionen:

- *beliebige Verteilungsfunktionen lassen sich durch (0,1)-gleichverteilte Verteilungsfunktionen approximieren*

3. Modell ← → Berechnungsmodell

determiniert berechnete Folgen von (0,1)-Werten lassen sich als Approximationen
von Verteilungsfunktionen realer Zufallsgrößen verwenden

Zufallsgrößen

Zufallsgrößen:= zufällige Ereignisse --> Zahlen

reale Zufallsgrößen und ihre Verteilungsfunktionen

Diskrete Zufallsgrößen:= Größen, die endliche oder abzählbar-unendlich viele verschiedene Werte annehmen können

Beispiel:

- Auszählen der Stillstände einer Maschine während einer Werksschicht
- Registrierung der Anzahl von Gesprächen in einer Telefonvermittlung

Stetige Zufallsgrößen:= Größen, die jeden beliebigen Wert innerhalb eines Intervalls der Zahlengerade annehmen können

Beispiel:

- Durchmesser von Antriebswellen (nach Bearbeitung an einem Drehautomaten): alle Werte innerhalb eines vorgeschriebenen Toleranzbereiches

Verteilungsfunktion einer Zufallsgröße

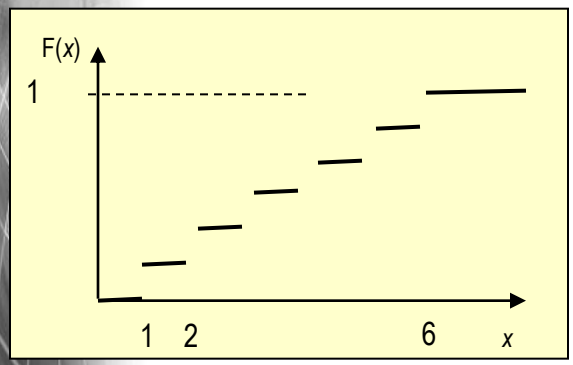
Charakterisierung einer Zufallsgröße X:

- X nimmt bei jedem Versuch zufällig einen bestimmten Wert an
- Werte genügen einer Verteilungsfunktion

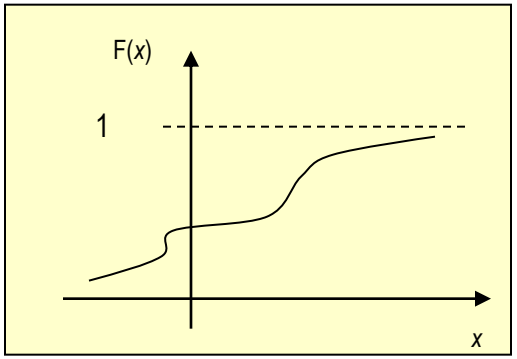
Verteilungsfunktion:= $F_X(x) = P(X \leq x)$,
der Wert von F_X (Verteilungsfunktion der Größe X) ist an der Stelle x ist **gleich der Wahrscheinlichkeit**, dass X einen Wert unterhalb von *x* annimmt.

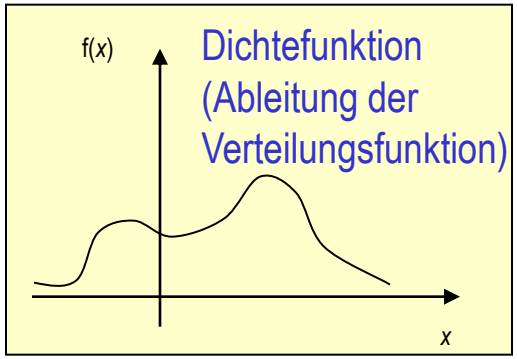
x durchläuft alle Werte der reellen Zahlengerade

diskret



kontinuierlich





Verteilungsfunktion als kumulative Dichtefunktion

Diskrete Zufallsgrößen

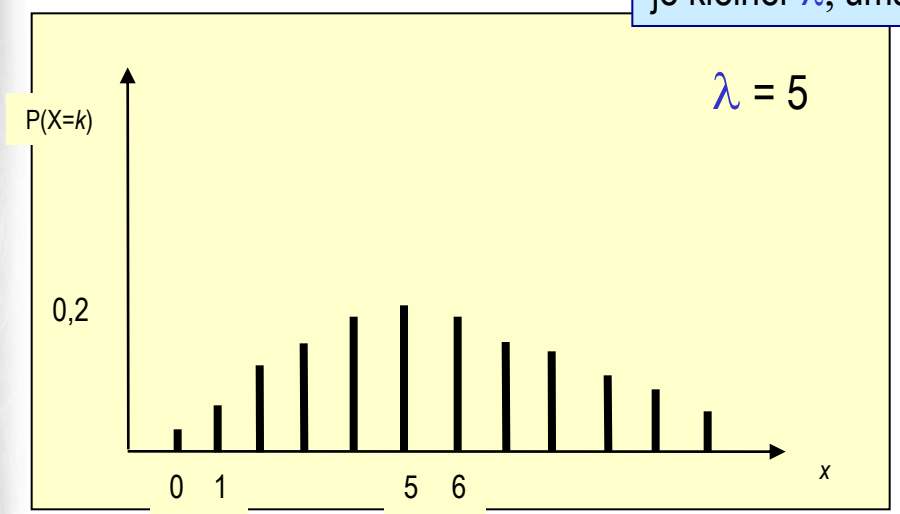
Beispiel einer diskreten Zufallsgröße X:

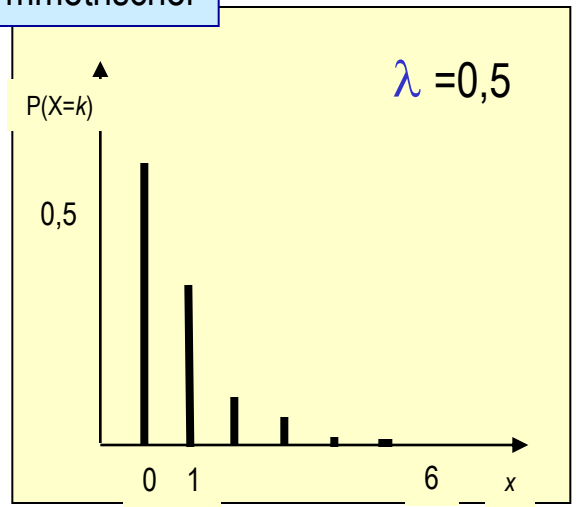
- X beschreibt Ereignisregistrierungen (Anzahl) in einem bestimmten Zeitbereich (X nimmt bei jedem Versuch zufällig einen best. Wert an)
- Werte genügen einem gleichen Typ von Verteilungsfunktion (**Poisson**)
- λ ist die mittlere Anzahl

Ereignisse

- Anzahl beobachteter Sternschnuppen in einer bestimmter Zeitspanne
- Anzahl registrierter Telefonanrufe in einer best. Zeitspanne
- Anzahl von Ankünften von Kunden einer Service-Einrichtung in best. Zeitspanne

je kleiner λ, umso unsymmetrischer





Verteilungsfunktion einer diskreten Zufallsgrößen

Poisson-Verteilung:

beschreibt Ereignisregistrierungen (Anzahl)
in einem bestimmten Zeitbereich

Verteilungsfunktion:= $F_X(x) = \sum_{k<x} P(X=k) = \sum_{k<x} \lambda^k/k! * e^{-\lambda}$,

falls x>0, sonst 0

$\mu = \lambda$, Erwartungswert
$\sigma^2 = \lambda$, Streuung

Wichtig: Poisson-Verteilung steht im Zusammenhang
mit der Exponentialverteilung:

Sei X poisson-verteilte Zufallsgröße mit Erwartungswert λ,
dann ist die Zwischenankunftszeit zweier aufeinanderfolgender Ereignisse
exponential-verteilt mit Erwartungswert $1/\lambda$

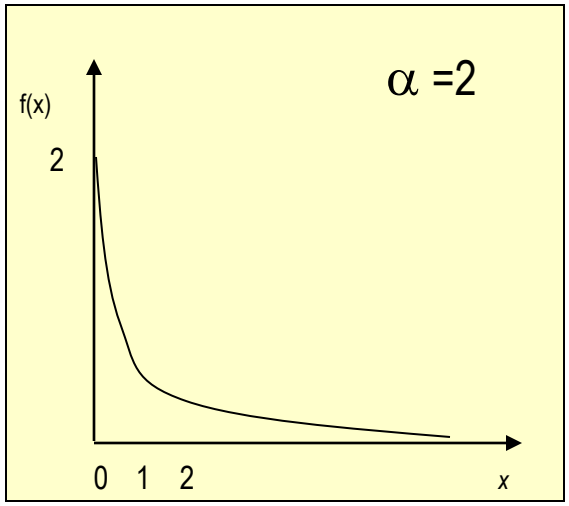
Stetige Zufallsgrößen

Beispiel einer stetigen Zufallsgröße X:

- X nimmt bei jedem Versuch zufällig einen bestimmten Wert an
- Werte genügen einer (negativen) Exponential-Verteilungsfunktion

Dichtefunktion $f(x) = \alpha e^{-\alpha x}$, falls $x>=0$, sonst 0

Verteilungsfunktion $F(x) = 1 - e^{-\alpha x}$, falls $x>=0$, sonst 0



- Zeitabstände zwischen Ankunftsereignissen von Ringstapeln in einer Schicht
- Dauer von Telefongesprächen
- Lebensdauer von Lebewesen, Maschinen, ...

$\mu = 1/\alpha$

$\sigma^2 = 1/\alpha^2$

5. ODEMx-Modul Random

1. Charakterisierung von Zufallsgrößen
2. Approximation von Zufallszahlen
3. ODEMx- Zufallszahlengeneratoren (Übersicht)
4. Einstellung von Startwerten
5. Protokollierung
6. Berechnung von Zufallszahlen ausgewählter Verteilungen

Pseudo-Zufallszahlen

Warum sind reproduzierbare Zufallszahlen enorm wichtig für die Simulation?

Iterationsverfahren

x_0 – beliebig aus [1, 10], z.B.: 2

$x_{k+1} \equiv 2 * x_k$ **mod** 11 (k= 0, 1, 2, …)

➔ Zahlenfolge mit Periode p:
x_0= 4, 8, 5, 10, 9, 7, 3, 6, 1, 2, 4, … (Wiederholung)

In Abhängigkeit vom Startwert

- ergibt sich eine periodische Folge von determinierten Zahlen, die als Zufallszahlenfolge interpretiert werden kann

Ein einfacher Pseudozufallszahlengenerator

```
class Random {
    int u;
    Random( int uu)  u(uu) {
            if u < 1 or u>10 { error(…);  …};
    }
    double next () {
            u:= 2*u;
            if u>11 { u= u-11; };
            next= double(u) / 11.0;
    }
}
```

Warum sind individuelle Startwerte der Generatoren wichtig für die Simulation?

Individuelle Startwerte: eigene Folgen von Zufallszahlen

```
Random * s1, * s2, *s3;
s1= new Random(2); s2= new Random(9); s3 = new Random(7);
```

```
s1->next();   // .636, .273, .545, …
s2->next();  // .364, .727, .455, …
s3->next();  // .273, .545, .091, …
```

Linearer Generator

Generator für gleichverteilte 26-Bit-Zufallswerte

$q = 67.099.547 = 2^{26} - 1$

Kongruenzmethode

- multiplikative Variante (Iterationsverfahren mit Startwert x_0)

$x_{j+1} \equiv k\, x_j$ **mod** q

$x_{j+1} \equiv 8192\, x_j$ **mod** $67.099.547 \quad (j = 0, 1, 2, 3, \ldots)$

➔ Zahlenfolge mit **sehr großer** Periode $p = 67.099.546$:

$x_0,\ x_1,\ x_2, \ldots,\ x_{p-1},\ x_p = x_0,\ x_1,\ x_2, \ldots$

Algorithmus für Startwerte (unabhängiger) Generatoren

$u_0 = 907$ (erster Startwert eines Generators)

$u_{k+1} \equiv 36.855 * u_k$ **mod** $67.099.547$

➔ 500 Generatoren für unabhängige Zahlenfolgen der Länge 120.000 (ohne dass eine Folge, in die andere „hineinläuft“)