

Eulersches Polygonzugverfahr en mit Java



by *Markus Lelie*

1

Einleitende Erläuterungen

Print

Open / Close

Vorwort

Open / Close

Ziel dieser Facharbeit ist es die Möglichkeit zu bieten in eine wie auch immer geartete Kombination aus Java und Mathematica eine Funktion eingeben zu können, deren Integralfunktion dann annäherungsweise mithilfe des Eulerschen Polygonzugverfahrens geplottet wird. Besonders interessant ist dieses Verfahren im Zusammenhang mit Funktionen, für die keine integralfreie Darstellung existiert.

Im Zuge der Realisierung war es erforderlich, mich näher mit bestimmten Themenbereichen zu beschäftigen, um die Kenntnisse zu erlangen, die im Endeffekt erforderlich waren.

Im Folgenden werde ich nun die nötigen Grundlagen ansatzweise beschreiben und dabei auf mir wichtig erscheinende Bereiche näher eingehen. Zudem werde ich meine Lösung des Problems erläutern, wobei beispielsweise auftretende Schwierigkeiten näherer Betrachtung unterzogen werden sollen.

Hinweis: In diesem Notebook sind der eigentliche Text der Facharbeit und der "Quellcode" vermischt worden. Bedingt durch Mathematicas Eigenarten könnten die Formatierungen hier also zu kurz kommen (z.B. waren nach dem Einfügen der ursprünglichen Texte in Mathematica diverse auf Satzzeichen folgende Leerzeichen und Zeilenumbrüche verschwunden).

Das Eulersche Polygonzugverfahren

Open / Close

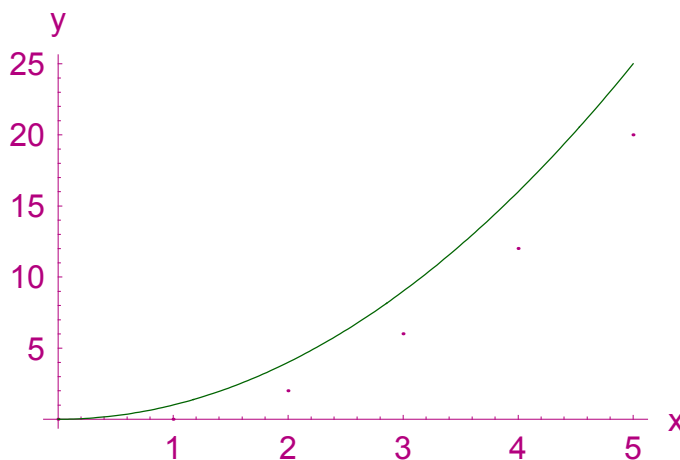
Das sog. Eulersche Polygonzugverfahren ist ein mathematisches Verfahren, mit dem sich die Integralfunktion einer gegebenen Funktion annäherungsweise bestimmen lässt, sofern von der gesuchten Integralfunktion die erste Ableitung, sowie der Funktionswert an einer Stelle bekannt sind.

Eine andere, weniger gängige Bezeichnung für dieses Verfahren (ILA = iterative lineare Approximation), lässt Schlüsse auf die Funktionsweise des selbigen zu:

Indem man, beginnend von der bekannten Stützstelle, also derjenigen Stelle, deren Funktionswert gegeben ist, iterativ Tangenten mithilfe der bekannten Ableitung als Maß für die Steigung bildet, wobei der zuletzt gewonnene Punkt wiederum als Stützstelle dient, erhält man bei einer (zumindest scheinbar) gegen unendlich strebenden Anzahl von Schritten einen annäherungsweisen Graphen der Integralfunktion.

Die Genauigkeit des entstehenden Graphen ist, wie aus Abbildung 1 gut ersichtlich ist (Punkte=Approximation, durchgezogene Kurve=Plot der gesuchten Fkt.), abhängig von der gewählten Schrittweite, d.h. von dem Abstand zwischen den einzelnen Stützstellen. Hier wurde eine Schrittweite von 1 auf die erste Ableitung der Funktion $f(x)=x^2$ angewandt, was im Vergleich zu der "richtigen" Funktion noch zu einem ungenauen Ergebnis führt.

Bei der Umsetzung des Verfahrens habe ich vereinfachend vorausgesetzt, dass der bekannte Funktionswert, der als Stützstelle dient zu der unteren Grenze der Zeichnung gehört.



Mathematica und JAVA

Open / Close

Um die Programmiersprache JAVA zusammen mit Mathematica zu nutzen, wird ein Mathematica-Addon namens JLink benötigt, welches sozusagen zwischen dem Java-Interpreter und Mathematica vermittelt.

Grundsätzlich gibt es zwei verschiedene Möglichkeiten, um beide Komponenten gemeinsam zu nutzen. Die Wahl der zu nutzenden Variante hängt davon ab, zu welchem Zweck Java und Mathematica im Rahmen des zu lösenden Problems eingesetzt werden sollen.

- Die eine Möglichkeit besteht darin eine Java-Anwendung zu schreiben, die Mathematica für nötige Berechnungen nutzt. Sie greift also sozusagen als vollkommen eigenständiges Programm von außen auf den Mathematica Kernel zu.
- Bei der anderen Methode wird der benötigte Java Quellcode in ein Mathematica-Notebook integriert, d.h. zusätzlich zu den Mathematica-eigenen Befehlen stehen sämtliche Java-Klassen zur Verfügung (sowohl diejenigen, die standardmäßig zur Java-Umgebung gehören, als auch spezielle, zu JLink gehörende Klassen). Zudem besteht die Möglichkeit, eigene Javaklassen einzubinden und zu nutzen.

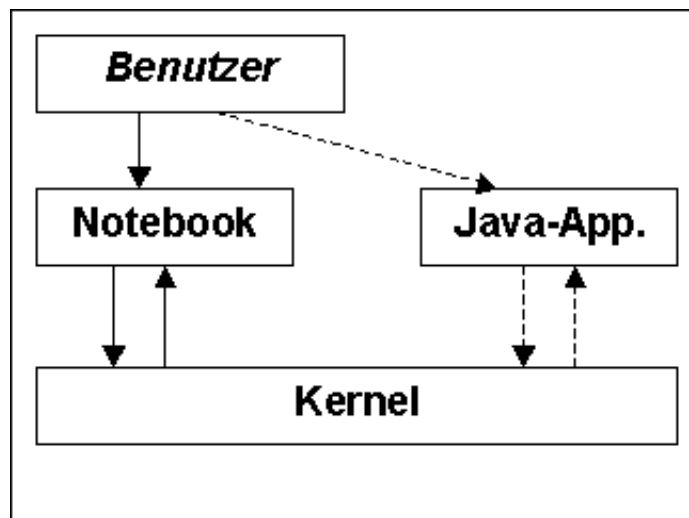
Die zuletzt beschriebene Variante eignet sich, um praktisch ein weiteres Frontend für Mathematica zu schaffen, welches entweder selbstständig, also ohne auf ein Mathematicanotebook zurückzugreifen, arbeiten kann, oder auch eine Erweiterung der normalen Fähigkeiten des Frontends darstellen kann.

Zum besseren Verständnis der folgenden Erläuterung der zweiten Variante ist hier ein kurzer Einschub über die die generelle Funktionsweise von Mathematica und JLink vonnöten:

Das Programm Mathematica besteht, grob vereinfacht, aus zwei Komponenten. Dem Kernel, der die eigentlichen Funktionen beherrscht, also für sämtliche Berechnungen und mathematischen Operationen zuständig ist, und dem Frontend – für den Benutzer sichtbar als Notebook – welches für die Eingaben zuständig ist, bzw. dafür, die Eingaben zu formatieren, an den Kernel weiterzuleiten und dem Benutzer die resultierenden Ausgaben zu präsentieren.

Dieser modulare Aufbau macht es im Übrigen möglich, beispielsweise den Kernel auf einem leistungsstarken Server in einem Netzwerk zu betreiben und auf diversen Clients mit Verbindung zum Server nur ein Frontend laufen zu lassen.

Dieses greift dann für die Berechnungen auf den auf dem Server laufenden Kernel zurück.



Soll nun unter Verwendung von Java ein Frontend für ein Mathematica Notebook geschaffen werden, das beispielsweise bestimmte, vom Benutzer zu machende Eingaben ermittelt, kommt Variante 2 zum Zuge.

Im Normalfall ist es jedoch eine Eigenschaft des Kernels, stets von einer Stelle Eingaben entgegen zu nehmen, typischerweise ist diese Stelle das Mathematica-Frontend/Notebook.

Will man also von einem durch Java erzeugten Fenster aus Befehle an den Kernel schicken, um die Ergebnisse zu erlangen, so muss zunächst dafür gesorgt werden, dass der Kernel das Fenster überhaupt „erhört“. Zu diesem Zweck könnte die gesamte Aufmerksamkeit des Kernels auf das Java-Fenster konzentriert werden, sodass andere Frontends keine Chance hätten „erhört“ zu werden. Dies hätte zur Folge, dass Eingaben in ein Notebook zur selben Zeit unbearbeitet blieben. Grundsätzlich eignet sich diese Kernel-Frontend-Beziehung (modal) eher für Eingabefenster mit Dialogcharakter, d.h. solche, die einmal Eingaben erwarten, anschließend geschlossen werden und keine Einfluss mehr auf das Geschehen haben.

Soll, wie in diesem Fall sinnvoll, ein Fenster als Frontend geschaffen werden, welches zunächst Eingaben entgegennimmt und anschließend entsprechende Ausgaben präsentiert, also Ergebnisse vom Kernel empfängt und eventuell auch eine erneute Ausführung mit angepassten Parametern erlaubt, so eignet sich eher eine andere Lösung.

Eine bei der dem Kernel mitgeteilt wird, dass er quasi mehreren Stellen gleichzeitig zur Verfügung zu stehen hat (modeless). Der Kernel teilt seine Aufmerksamkeit (Abb. 2), was sich in der Praxis übrigens durch das Wort sharing vor dem

Eingabeprompt in einem Notebook zeigt.

2

Die Umsetzung

Print

Open/Close

Erläuterungen zur Umsetzung

Open / Close

Für die Umsetzung habe ich mich aufgrund der oben genannten Gründe für einen „geteilten“ Kernel entschieden. Aus einem Mathematica-Notebook heraus wird ein Java-Fenster erzeugt, welches einen Button, eine Zeichenfläche, sowie ein Textfeld enthält.

Ursprünglich hatte ich vorgesehen, die Möglichkeit zu bieten, sämtliche erforderlichen Parameter, wie die Funktion, die Grenzen, die Stützstelle sowie die Schrittweite in dem als GUI (Graphical User Interface) fungierenden Fenster einzugeben.

Leider ist es mir jedoch bisher nicht gelungen, einen in ein Textfeld eingegebenen Funktionsterm als Funktion in Mathematica zu definieren. Lediglich Zahlenwerte konnten eingelesen und verarbeitet werden.

Aus diesem Grund habe ich mich entschieden, alle Parameter bis auf die Schrittweite zentral im Notebook zu definieren. Somit besteht die Möglichkeit, die Approximation mehrfach mit verschiedenen Schrittlängen durchzuführen und so durch Verkleinerung der Schrittlänge auf Kosten der Rechenzeit einen genaueren Graphen zu erhalten.

```

1. xMi = -5; (*linke Grenze der Zeichnung*)
   xMa = 5; (*rechte Grenze der Zeichnung*)
   schrWeite = 0.01; (*Schrittweite/Abstand zwischen den
   einzelnen Stützpunkten*)
   SYB = -125; (* Funktionswert der gesuchten Integralfkt.
   an der Stelle xMi also F[xMi]*)
   f[x_] := 3x^2 (* Erste Ableitung der gesuchten Fkt.*)

```

Die an dieser Stelle eingegebene Schrittweite wird beim Aufruf des Fensters als Startwert verwendet.

Das entstandene Notebook ist in vier Bereiche geteilt.

Zunächst muss der JAVA-Interpreter geladen und über JLink mit Mathematica verknüpft werden, was durch die folgenden Zeile geschieht.

```
<<JLink`  
2. InstallJava[CommandLine->"E:\Program  
Files\jdk1.3.1\bin\java.exe"];
```

An dieser Stelle trat auch bereits das erste Problem auf, da InstallJava nicht in der Lage war, meinen Java-Interpreter zu finden.

Abhilfe schafft das Attribut CommandLine, mit dem der genaue Pfad zu dem zu verwendenden Interpreter festgelegt wird.

Um dieses Notebook an anderer Stelle auszuführen, muss der Parameter CommandLine entweder entfernt, oder (für den Fall, dass der Interpreter nicht gefunden wird) angepasst werden.

Als nächstes folgt der Teil, in dem die später benötigten Funktionen bzw. Methoden definiert werden. Die Namen sollten selbsterklärend sein:

- FensterErz[] erzeugt ein neues Fenster vom Typ com.wolfram.jlink.MathFrame und bestückt dieses mit den nötigen Objekten.
- Die Methode Ausrechnen[] wird dann aufgerufen, wenn der Button gedrückt wurde und enthält das eigentliche Kernstück des Programms. Es wird eine Tabelle mit den zu plottenden Funktionswerten erzeugt, die anschließend als Graph ausgegeben wird.
- Es folgt die Methode EndFunct[], die dafür sorgt, dass die global deklarierten Objekte nach dem Schließen des Fensters wieder entfernt werden. Somit ist eine erneute Ausführung möglich, ohne vorher durch das Beenden des Java-Interpreters und u.U. auch des Kernels alle Objekte zwangsmäßig freizugeben.
- An letzter Stelle folgt die Methode StartIt[], deren einzige Aufgabe es ist, die FensterErz[]-Methode aufzurufen.

Die beiden bisher beschriebenen Komponenten sind in Mathematica-Zellen verpackt, deren InitializationCell-Eigenschaft auf True gesetzt wurde, was dazu führt, dass dem Benutzer bei Evaluierung einer beliebigen Zelle des Notebooks die Option geboten wird, alle Initialisierungs-Zellen automatisch auszuführen, bevor etwas anderes passiert.

Somit ist sichergestellt, dass die als nächstes folgende Zelle, deren Ausführung den Start des „Programms“ bewirkt, nur dann evaluiert wird, wenn dem Kernel vorher alle Methoden bekannt gemacht und Java gestartet wurde.

Soll schließlich die Anwendung gestartet werden, so muss die im Notebook mit

Start betitelte Zelle ausgeführt werden. Die Arbeitsweise Javas macht es erforderlich, die Zeichenfläche sowie das Textfeld global zu deklarieren.

Dies rührt daher, dass Objekte in Java bestimmte Gültigkeitsbereiche haben. Ein in einer bestimmten Methode deklariertes Objekt kann zwar aus der selbigen heraus problemlos manipuliert werden, ein Zugriff von außen ist jedoch normalerweise nicht möglich.

Da die Zeichenfläche und das Textfeld zwar in unser Fenster gehören und diesem in der Methode FensterErz[] zugeordnet werden, trotzdem jedoch die Werte in der Methode Ausrechnen[] abgerufen, bzw. im Falle der Zeichenfläche verändert werden müssen, ist die Notwendigkeit gegeben, diese Objekte global zu deklarieren.

```
3.  tbFkt=JavaNew["java.awt.TextField"];
    zeich=JavaNew["com.wolfram.jlink.MathCanvas"];
```

Das heißt natürlich auch, dass diese Objekte bestehen bleiben, wenn das Fenster geschlossen wird und alle anderen Objekte und Variablen freigegeben werden, da sie in der FensterErz[]-Methode innerhalb eines JavaBlocks lokal definiert waren:

```
4.  FensterErz[] := JavaBlock[
    Module[{wnd, listener, windowListener, cmdPlot},
        ...
    ]
]
```

Wie weiter oben bereits erwähnt wurde, müssen die globalen Objekte also nach dem Schließen des Fensters gesondert „entsorgt“ werden.

Hierzu wird dem Fenster ein sog. listener hinzugefügt.

Dies ist eine Klasse, die vereinfacht gesagt, auf bestimmte Ereignisse wartet, die bei einem Objekt auftreten können, z.B. das Schließen eines Fensters, und bei deren Auftreten bestimmte Reaktionen auslöst, in diesem Fall eben die Freigabe der global definierten Objekte durch den Aufruf der Methode EndFuncnt[].

```
5.  listener = JavaNew["com.wolfram.jlink.MathActionListener"];
    ...
    wnd@addWindowListener>windowListener];
    windowListener@
    setHandler["windowClosing", "EndFuncnt[]&"];
```

Auch für die Funktion des Buttons zum Plotten ist ein Listener erforderlich. Dieser ruft bei auftreten des ActionPerformed-Ereignisses die Ausrechnen[]-Methode auf,

die ich etwas genauer beschreiben will, da in ihr die meiste Arbeit und der meiste Ärger stecken, auch wenn sie auf den ersten Blick ganz harmlos erscheint.

```

6. (* Berechnung der Punkte *)
    tmpVal = {xMi, SYB};
    AppendTo[werte, tmpVal];
    SY = SYB;

    For[xL = xMi, xL < xMa, xL += schrWeite,
        SY = SY + f[xL];
        tmpVal = {xL + schrWeite, (SY) * schrWeite + SYB};
        AppendTo[werte, tmpVal];
    ]

zeich@setMathCommand[
    "ListPlot[werte, PlotStyle->{Blue,PointSize[.005]}]";
zeich@repaintNow[];

```

Zunächst wird das erste, bereits bekannte Wertepaar gespeichert. Es hat sich herausgestellt, dass es einfacher ist den Mathematica-Datentyp list zur Zwischenspeicherung zu verwenden, als zu diesem Zweck ein Java-Array zu definieren. Da die Schrittweite variabel sein soll, reicht es nicht, nur eine list für die Funktionswerte zu erstellen. Die zum Zeichnen verwendete Funktion ListPlot würde nämlich die Liste als Ansammlung von Funktionswerten zugehörig zu x-Werten von 1 bis n interpretieren, deren Abstand 1 beträgt. Somit wäre der Graph extrem verfälscht.

Vermieden wird dieser Effekt durch die Erzeugung einer Liste von Koordinaten/Wertepaaren – praktisch n Listen mit jeweils zwei Werten (x- und y-Koordinate) vereint in einer Liste – die schließlich der ListPlot-Funktion übergeben wird.

Die eigentliche Berechnung der zu zeichnenden Punkte erfolgt – nach dem eingangs, bei der Erläuterung des ILA-Verfahrens beschriebenen Schema – in der For-Schleife.

Wichtig ist, dass die Konstante SYB (der y-Wert der ersten Stützstelle) zu jedem neu gewonnenen Funktionswert addiert wird, andernfalls ist der entstehende Graph nämlich um diesen Betrag nach oben oder unten verschoben. Die Erklärung liefert der bei der Integration auftretende konstante Summand, der nur so mit eingehen kann.

Die vorletzte Zeile sorgt für die Zeichnung des Graphen, wobei das Attribut

PlotStyle nur bei gleichzeitiger Verwendung von MDTools Wirkung zeigt, ansonsten meiner Erfahrung nach aber auch keinen Schaden verursacht. Der Aufruf von `repaintNow[]` stellt sicher, dass der ListPlot-Befehl auch umgesetzt wird, es sollte aber auch ohne diesen funktionieren.

Zu erwähnen bleibt jetzt eigentlich nur noch der letzte, mit Spuren beseitigen betitelt Punkt. Die Ausführung dieser Zelle schließt den Java-Interpreter und beendet die „Aufteilung“ des Kernels, nachdem die vorher definierten Funktionen wieder entfernt wurden.

Der vollständige Quelltext des Notebooks befindet sich sowohl in maschinen- als auch für Menschen lesbarer Form im Anhang.

Literaturverzeichnis

Open / Close

- Flanagan, David, O'Reilly, JAVA in a nutshell, Deutsche Ausgabe für Java 1.2 und 1.3, 2001, 2. korrigierter Nachdruck der 3. Auflage, o.O.
- Donnerberg, Mersch, Podewin, Artikel Integration einer Funktion durch iterative lineare Approximation (ILA) mit Einsatz eines CAS, Herkunft, Ort und Datum unbekannt; Artikel siehe Anlage
- Todd Gayley, J/Link User Guide, Version 2.0, Juni 2002, o.O.

3

Quelltext

Print

Open / Close

Initialisierung/Methoden

Open / Close

```
Input > (*<<JLink`
InstallJava[
  CommandLine->"E:\Program Files\jdk1.3.1\bin\java.exe"];*)
```

```
Input > (*Geänderter Pfad*)
<< JLink`
InstallJava[CommandLine -> "C:\j2sdk1.4.1_02\bin\java.exe"];
```

```

FensterErz[] := JavaBlock[
  Module[{wnd, listener, windowListener, cmdPlot},

    (*Kernel mit Frontend "teilen"*)
      ShareKernel[];

    (* benötigte Objekte erzeugen *)
      wnd = JavaNew["com.wolfram.jlink.MathFrame", "ILA"];
      listener =
      JavaNew["com.wolfram.jlink.MathActionListener"];
      windowListener = JavaNew[
      "com.wolfram.jlink.MathWindowListener"];
      cmdPlot = JavaNew["java.awt.Button"];

    (* Neues Fenster erzeugen und anpassen *)
      wnd@setLayout[JavaNew["java.awt.BorderLayout"]];
      wnd@setSize[1000, 400];
      wnd@addWindowListener>windowListener];
      windowListener@
      setHandler["windowClosing", "EndFunc[]&"];

    (* Eigenschaften des Grafikbereichs setzen *)
      (*zeich@setImageType[MathCanvas`TYPESET];
      zeich@setMathCommand["Sqrt[25]"];
      *)
      zeich@setImageType[MathCanvas`GRAPHICS];
      zeich@setUsesFE[True];

    (* Eigenschaften des Buttons festlegen *)
      cmdPlot@setLabel["=>>"];
      cmdPlot@addActionListener[listener];
      listener@setHandler["actionPerformed", "Ausrechnen"];

    (* Objekte einfügen und positionieren *)
      wnd@add[cmdPlot, ByRef[BorderLayout`WEST]];
      wnd@add[tbFkt, ByRef[BorderLayout`NORTH]];
      wnd@add[zeich, ByRef[BorderLayout`CENTER]];

    tbFkt@setText[ToString[schrWeite]];
    (* Fenster anzeigen *)
      JavaShow[wnd];

  ]
]

```

```

Ausrechnen[_ , _] :=
Module[{inFkt, newV, xL},

(*eventuell vorhandene,
von vorherigen Berechnungen übrige Werte löschen*)
werte = {};
xAchse = {};
(*Einlesen der gewählten Schrittweite *)
schrWeite = tbFkt@getText[];
schrWeite = ToExpression[schrWeite];

(*Der gescheiterte Versuch,
die Funktion über das Eingabefeld zu erhalten*)
(*SYB=tbSY@getText[];
SYB = ToExpression[SYB];
Clear[f];
f[x_] := inFkt;
*)

(* Berechnung der Punkte *)
tmpVal = {xMi, SYB};
AppendTo[werte, tmpVal];
SY = SYB;

For[xL = xMi, xL < xMa, xL += schrWeite,
SY = SY + f[xL];
tmpVal = {xL + schrWeite, (SY) * schrWeite + SYB};
AppendTo[werte, tmpVal];
]

zeich@setMathCommand[
"ListPlot[werte, PlotStyle->{Blue,PointSize[.005]}]"];
zeich@repaintNow[];

]
Null

EndFunc[] := JavaBlock[
ReleaseObject[tbFkt];
ReleaseObject[zeich];
]

```

Input >

```

StartIt[] :=
  Input >  JavaBlock[
           FensterErz[];
           ]

```

Start

Open / Close

```

Clear[f];

xMi = -5; (*linke Grenze der Zeichnung*)
xMa = 5; (*rechte Grenze der Zeichnung*)
schrWeite = 0.01;
(*Schrittweite/Abstand zwischen den einzelnen Stützpunkten*)
SYB = -125; (* Funktionswert der gesuchten Integralfkt.
an der Stelle xMi also F[xMi]*)
Input >  f[x_] := 3 x^2 (* Erste Ableitung der gesuchten Fkt.*)

(* benötigte globale Objekte erzeugen *)
tbFkt = JavaNew["java.awt.TextField"];
zeich = JavaNew["com.wolfram.jlink.MathCanvas"];

StartIt[];

```

"Spuren beseitigen"

Open / Close

```

Clear[FensterErz, Ausrechnen, EndFunct, StartIt, f];
Input >  UnshareKernel[];
        UninstallJava[];

```

[New Chapter](#)

[Cut last Chapter](#)

(c) by Markus Lelie