Standard

zur Komprimierung von Zahlenreihen

http://www.dynamo-software.de/

2017-12-17. Jens-Erich Lange, Fa. Dynamo Software
Tel. (04554 9997610) E-Mail: info@dynamo-software.de

Einleitung

Ab dem Jahr 2000 wurde ich immer wieder mit unterschiedlichen Messreihen aus der Industrie und aus der Medizintechnik konfrontiert. Im Jahr 2008 ging ich dann der Frage nach, ob es möglich wäre, diese Messreihen auch in komprimierter Form abzulegen. Ich war verwundert darüber, dass es dazu offenbar noch keine standardisierten Verfahren gab:

http://de.comp.lang.delphi.misc.narkive.com/TiEiCHq7/messwertkurven-optimal-komprimieren

Im Jahr 2012 gab es dann endlich den Startschuss, um nach einer eigenen Lösung zu suchen. Aus den damals entstandenen Routinen „PackIntegers“ und „UnpackIntegers“ entstand nun fünf Jahre später eine vollständig implementierte Klasse, die beliebige Zahlenreihen im Fließkommaformat packt und entpackt.

Datenreduktion durch Komprimierung

Wir betrachten Messwertverläufe, wie sie bei Verschraubungen oder Einpressvorgängen entstehen. Diese Messkurven haben typischerweise zwei Achsen (=Kanäle oder Dimensionen), zum Beispiel „Winkel“ = X und „Drehmoment“ = Y. Manchmal kommen weitere Kanäle (Datenreihen) hinzu.

Betrachten wir eine Messkurve mit den drei Kanälen „X“, „Y“ und „Z“. Aus den Kanälen und den einzelnen Messwerten (Samples) spannt sich ein zweidimensionales Werte-Array auf. Wie wird dieses Werte-Array günstigerweise abgelegt (serialisiert)?

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**…**
X	a								
Y	b								
Z	c								

Grundsätzlich besteht die Möglichkeit die Datenreihen als „Samples“ zu organisieren. Dabei werden wie in der obigen Abbildung zu einem gegebenen Messpunkt alle Kanäle zusammen angegeben. Dies erscheint zunächst als die natürliche und auch als die einfachere Herangehensweise.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**…**
X	a	b	c	d	e	f	g	h	…
Y									
Z									

Andererseits können die Daten des Arrays auch über ihren „Run“, also die Datenreihe eines einzelnen Kanals organisiert werden. Diese Herangehensweise ist in der obigen Abbildung angedeutet und empfiehlt sich aus mehreren Gründen:

- Falls es nur einen einzigen Kanal gibt, erscheint diese Methode logischer und einfacher.
- Falls Daten in lesbarer Form abgelegt werden sollen, werden die Wiederholungen pro Sample vermieden. Das Array hat in der Nord-Süd Ausrichtung vielleicht 10 Einträge. In der Ost-West Ausrichtung jedoch leicht einige 1000.
- Diese Methode bietet eine sehr gute Möglichkeit zur Datenreduktion durch Komprimierung und ist daher entscheidend für die nachfolgend vorgestellten Routinen.

Die wichtigste Maßnahme ist es, die Differenzen zwischen den einzelnen Elementen der Messreihe zu bilden. Dabei zeigt sich folgendes:

- Je näher ein Wert an Null liegt (positiv oder negativ) um so häufiger kommt er vor.
- Ein einzelner Wert wiederholt sich oft.

Diese Gegebenheiten macht sich der Algorithmus zu nutze. Bestehen die Deltas dagegen aus Zufallszahlen, so versagt der Algorithmus ebenso wie auch der ZIP Algorithmus versagt.

PackIntegers / UnpackIntegers

Für die Datenkomprimierung schlage ich einen Basis-Algorithmus vor, der in der Lage ist, ein Array von Integerwerten zu komprimieren/dekomprimieren. Als Basis dienen Integerwerte, die jedoch durch eine angenommene Kommastelle als Fließkommazahlen gelesen werden können. Der Algorithmus zeichnet sich durch folgende Eigenschaften aus:

- Hohe Kompressionsrate, meistens besser als das ZIP-Verfahren.
- Sehr einfach zu implementieren.
- Unabhängig vom Integerformat (Little/Big Endian)
- Funktioniert mit 32-Bit und 64-Bit Integern

Vorgehensweise beim komprimieren:

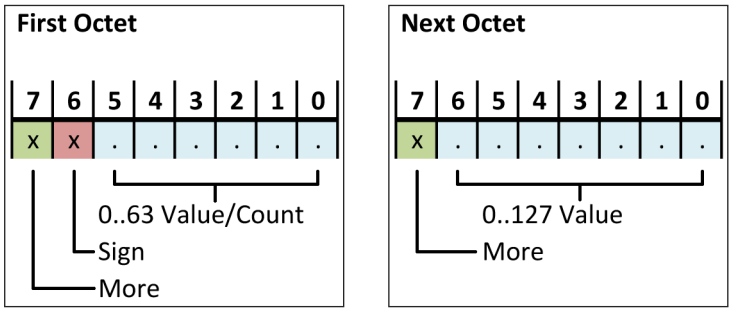
Zu jedem Integer-Wert wird als erstes der Differenzwert zu seinem Vorgänger ermittelt (Startwert = 0). Mit diesem Differenzwert wird fortgefahren.

Als nächstes wird der zu bearbeitende Differenzwert als Absolutwert kopiert, das Vorzeichen wird also abgetrennt und als Bit 6 im ersten (und vielleicht einzigem) Byte gespeichert.

Sodann werden die nachfolgenden Differenzwerte untersucht. Sind sie identisch, wird eine Wiederholzahl bis maximal 63 gezählt.

Ist die Wiederholzahl Null und der Absolutwert kleiner als 64, so wird dieser in den Bits 0…5 des ersten Bytes gespeichert. Das Bit 7 bleibt leer und die Bearbeitung dieses Wertes ist fertig. Diese Bedingung enthält eine redundante Möglichkeit, nämlich den Differenzwert „-0“. Dieser spezielle Wert ($40) dient in späteren Versionen als Escape-Prozedur.

Ist die Wiederholzahl > Null oder der Absolutwert > 63, so wird die Wiederholzahl in den Bits 0…5 des ersten Bytes gespeichert. Das Bit 7 des ersten Bytes wird gesetzt (das heißt, es folgt mindestens ein weiteres Byte) und mit der Bearbeitung wird fortgefahren. Für das zweite und die folgenden Bytes besteht folgende Vorschrift: Der Betrag der Modulo-Division von Absolutwert durch 128 wird in die Bits 0…6 eingetragen. Der Absolutwert wird ganzzahlig durch 128 dividiert. Ist der Absolutwert danach größer als Null, so wird das Bit 7 gesetzt und mit einem weiteren Byte fortgefahren. Der letzte Schritt wird so oft wiederholt, bis der Absolutwert in 7-Bit Blöcke vollständig aufgeteilt ist.



Die Dekomprimierung erfolgt entsprechend in umgekehrter Weise.

Quellcode

Der hier dargestellte Quellcode ist für das Verständnis des Algorithmus und für die Portierung in andere Hochsprachen gedacht. Für eine optimale Performance sollten die folgenden Prozeduren in Assembler kodiert werden.

```
procedure PackIntegers(Values: Integers; var Output: string);
var
  Index: integer;   //Actual index into the value array
  Delta: SINT;      //Difference between actual value and its predecessor
  Previous: SINT;   //Previous value
  Block: UINT;      //Absolute value of the actual delta value
  Count: word;      //Number of repeats (up to 63)
  Code: byte;       //actual coded octet
begin
  Output := '';
  Index := 0;
  Previous := 0;

  repeat
    Count := 0;
    Delta := Values[Index] - Previous; // Process the difference of this and previous value
    Block := abs(Delta); // absolute Value of Delta
    Previous := Values[Index];

    while (Count < 63) and (Index < High(Values)) and ((Values[Index + 1] - Previous) = Delta) do
    begin // Count up to 63 repeating deltas
      inc(Index);
      inc(Count);
      Previous := Values[Index];
    end;

    if (Count = 0) and (Block < 64) then Code := Block //The EASY way: Just one byte
      else Code := Count or $80; //Set Bit 7 because one more Bytes is following

    if (Delta < 0) then Code := Code or $40; // Set Bit 6 for minus sign
    Output := Output + char(Code);

    while (Code > $7F) do // At least one more Byte to go
    begin
      Code := Block mod 128;   // devide the remaining block into 7-Bit chunks
      Block := Block div 128;
      if (Block > 0) then Code := Code or $80; //Set Bit 7 because one more Byte is following
      Output := Output + char(Code);
    end;

    inc(Index);
  until (Index > High(Values));
end;
```

```
procedure UnpackIntegers(Input: string; var Values: Integers);
var
  i, j: integer;    //Loop variables
  Code: byte;       //Actual octet
  Sign: boolean;    //Sign of the actual value
  Count: word;      //Number of repeats (up to 63)
  Value: SINT;      //Actual value (including sign)
  Block: UINT;      //Actual value (absolute)
  Stack: string;    //Helper stack for collecting the 7-bit chunks of the block value
  Previous: SINT;   //Previous value
begin
  SetLength(Values, 0);
  i := 0;
  Previous := 0;
  while (i < length(Input)) do
  begin
    inc(i);
    Code := ord(Input[i]);

    if (Code = $40) then
    begin
      //ESCAPE Procedure
      Stack := '';
      repeat
        if (i < length(Input)) then
        begin
          inc(i);
          Code := ord(Input[i]);
          Stack := Stack + char(Code and $7F);
        end;
      until (Code < $80) or (i >= length(Input));
    end
    else begin
      //Regular Procedure
      Sign := (Code and $40) = $40;

      if (Code > $7F) and (i < length(Input)) then // More than one Byte is needed for coding
      begin
        Count := Code and $3F;
        Stack := '';
        repeat
          inc(i);
          Code := ord(Input[i]);
          Stack := char(Code and $7F) + Stack;
        until (Code < $80) or (i >= length(Input)); // Repeat until no more bytes left for coding

        Block := 0;
        for j := 1 to length(Stack) do Block := (Block * $80) + ord(Stack[j]);
      end
      else begin // EASY! Only one octet with 6 Bit value (+/- 63)
        Count := 0;
        Block := Code and $3F;
      end;

      Value := Block;
      if Sign then Value := -Value;

      for j := 0 to Count do //Output repeating values
      begin
        Previous := Previous + Value;
        SetLength(Values, length(Values) + 1);
        Values[High(Values)] := Previous;
      end;
    end;
  end;
end;
```

NumberPacker

Um aus den Basisprozeduren „PackIntegers“ und „UnpackIntegers“ einen vollständige Klasse zum packen und entpacken von Fließkommazahlen zu bauen, müssen folgende Erweiterungen vorgenommen werden:

Die Klasse enthält neben dem Array der Fließkommazahlen einen Wert „Digits“ (Nachkommastellen). Zu diesem Wert errechnet sich ein weiterer Wert „Factor“.

Digits = 3 Factor = 1000
Digits = 2 Factor = 100
Digits = 1 Factor = 10
Digits = 0 Factor = 1
Digits =-1 Factor = 0,1
Digits =-2 Factor = 0,01
Digits =-3 Factor = 0,001

Der Wert von Factor wird über die Potenz von Digits zur Basis 10 errechnet.

Damit die Klasse mit beliebigen Zahlen im Textformat oder Fließkommaformat „gefüttert“ werden kann, ohne dass die Anzahl der Nachkommastellen vorgegeben oder bekannt ist, muss eine Routine zum Erkennen der maximalen Nachkommastellen eingebaut werden.

Die Fließkommazahlen werden mit „Factor“ multipliziert und der resultierende Wert als Integerwert für die „PackIntegers“ Routine verwendet. Der Wert von „Digits“ wird als SignedByte den Binärdaten vorangestellt. Zusätzlich werden als Prüfmerkmal für das Datenformat die Zeichen „X“ und „1“ den Binärdaten vorangestellt.

Base64 Text

Sollen die Binärdaten als Text dargestellt werden, so sind sie durch die standardisierte „Base64“ Kodierung umzuwandeln.

https://de.wikipedia.org/wiki/Base64

Quellcode

Vollständiger Interface-Teil der Unit „NumberPack“ mit Deklaration der Klasse „TNumberPack“.

```
{*******************************************************}
{                                                       }
{       Unit "NumberPack"                               }
{                                                       }
{       Universal compression routine for numbers       }
{       2012, Initial Version 1.00, Jens-Erich Lange    }
{                                                       }
{       2017-08-17: Version 1.10, Jens-Erich Lange      }
{                                                       }
{*******************************************************}

unit NumberPack;

interface

const
  MAX_DIGIT = 9;

type
  {$IFDEF LOW_RESOLUTION}
    SINT = Longint;   //32 Bit Signed Integer
    UINT = Cardinal;  //32 Bit Unsinged Integer
    FLOAT = double;   //64 Bit precision Float
  {$ELSE}
    SINT = Int64;     //64 Bit Signed Integer
    UINT = UInt64;    //64 Bit Unsinged Integer
    FLOAT = extended; //80 Bit precision Float
  {$ENDIF}

  Integers = array of SINT;
  Numbers = array of FLOAT;


  TNumberPack = class(TObject)
  private
    fAutomatic: boolean;
    fDigits: integer;
    fFactor: Double;
    fValues: Integers;
    fFloats: Numbers;
    procedure FindDigits(S: string);
    function Append(Num: FLOAT): integer;
    function GetCount: integer;
    procedure SetDigits(Digits: integer);
    function GetText: string;
    procedure SetText(S: string);
  public
    constructor Create;
    destructor Destroy; override;
    procedure Clear;

    procedure Add(Value: FLOAT); overload;
    procedure Add(Value: string); overload;
    procedure AddArray(Values: Integers); overload;
    procedure AddArray(Values: Numbers); overload;

    function GetInt(Index: integer): SINT;
    function GetNum(Index: integer): FLOAT;
    function GetIntStr(Index: integer): string;
    function GetNumStr(Index: integer): string;

    property Count: Integer read GetCount;
    property Items[Index: Integer]: SINT read GetInt; default;
    property Digits: integer read fDigits write SetDigits;
    property Factor: Double read fFactor;

    property Text: string read GetText write SetText;
  end;


procedure PackIntegers(Values: Integers; var Output: string);
procedure UnpackIntegers(Input: string; var Values: Integers);
```

```
constructor TNumberPack.Create;
begin
  inherited Create;
  Clear;
end;
```

```
destructor TNumberPack.Destroy;
begin
  Clear;
  inherited Destroy;
end;
```

```
procedure TNumberPack.FindDigits(S: string);
var
  D,P: integer;
begin
  D := 0;
  while (length(S) > 0) and (S[length(S)] = '0') do
  begin
    delete(S, length(S), 1);
    dec(D);
  end;

  P := pos('.', S);
  if (P <> 0) then
  begin
    delete(S, 1, P);
    D := length(S);
  end;

  if (D > fDigits) then fDigits := D;
end;
```

```
function TNumberPack.Append(Num: FLOAT): integer;
begin
  SetLength(fFloats, length(fFloats) + 1);
  SetLength(fValues, length(fFloats));
  fFloats[High(fFloats)] := Num;
  Result := High(fFloats);
end;
```

```
function TNumberPack.GetCount: Integer;
begin
  Result := length(fValues);
end;
```

```
function TNumberPack.GetText: string;
var
  i: integer;
  D: Shortint; //Signed Byte
begin
  fFactor := IntPower(10, fDigits);
  for i := Low(fFloats) to High(fFloats) do
  begin
    try
      fValues[i] := round(fFloats[i] * fFactor);
    except
      // Something went terribly wrong here!
      // Happens only if Digits is out of range.
      fValues[i] := 0;
    end;
  end;
  PackIntegers(fValues, Result);
  D := fDigits;
  Result := 'X1' + char(D) + Result;
end;
```

```
procedure TNumberPack.SetText(S: string);
var D: Shortint; //Signed Byte
begin
  SetLength(fValues, 0);
  if (length(S) > 3) then
  begin
    if (S[1] = 'X') and (S[2] = '1') then
    begin
      move(S[3], D, 1);
      SetDigits(D);
      fFactor := IntPower(10, fDigits);
      delete(S, 1, 3);
      UnpackIntegers(S, fValues);
    end;
  end;
end;
```

```
procedure TNumberPack.SetDigits(Digits: integer);
begin
  fDigits := Digits;
  fAutomatic := false;
end;
```

```
procedure TNumberPack.Clear;
begin
  fAutomatic := true;
  fDigits := -MAX_DIGIT;
  fFactor := 0;
  SetLength(fFloats, 0);
  SetLength(fValues, 0);
end;
```

```
procedure TNumberPack.Add(Value: FLOAT);
var S: string;
begin
  if fAutomatic and (fDigits < MAX_DIGIT) then
  begin
    str(Value: 1: MAX_DIGIT, S);
    FindDigits(S);
  end;

  Append(Value);
end;
```

```
procedure TNumberPack.Add(Value: string);
var
  Num: FLOAT;
  Code: integer;
begin
  if (Value = '') then Value := '0';
  val(Value, Num, Code);
  if (Code = 0) then
  begin
    if fAutomatic and (fDigits < MAX_DIGIT) then FindDigits(Value);
    Append(Num);
  end;
end;
```

```
procedure TNumberPack.AddArray(Values: Integers);
var i: integer;
begin
  for i := low(Values) to high(Values) do Add(Values[i]);
end;
```

```
procedure TNumberPack.AddArray(Values: Numbers);
var i: integer;
begin
  for i := low(Values) to high(Values) do Add(Values[i]);
end;
```

```
function TNumberPack.GetInt(Index: Integer): SINT;
begin
  if (Index < 0) or (Index > High(fValues)) then Result := 0
  else Result := fValues[Index];
end;
```

```
function TNumberPack.GetNum(Index: integer): FLOAT;
begin
  try
    Result := GetInt(Index) / fFactor;
  except
    // Something went terribly wrong here!
    // Happens only if Digits is out of range/not set.
    Result := 0.0;
  end;
end;
```

```
function TNumberPack.GetIntStr(Index: integer): string;
begin
  str(GetInt(Index): 1, Result);
end;
```

```
function TNumberPack.GetNumStr(Index: integer): string;
begin
  str(GetNum(Index): fDigits, Result);
end;
```