

# fffg.spad

Martin Rubey

July 24, 2008

## Abstract

The packages defined in this file provide fast fraction free rational interpolation algorithms.

## Contents

|   |   |    |
|---|---|----|
| 1 | package FAMR2 FiniteAbelianMonoidRingFunctions2 | 1  |
| 2 | package FFFG FractionFreeFastGaussian           | 3  |
| 3 | package FFFGF FractionFreeFastGaussianFractions | 14 |
| 4 | package NEWTON NewtonInterpolation              | 17 |
| 5 | License   | 18 |

## 1 package FAMR2 FiniteAbelianMonoidRingFunctions2

```
<package FAMR2 FiniteAbelianMonoidRingFunctions2>≡
)abbrev package FAMR2 FiniteAbelianMonoidRingFunctions2
++ Author: Martin Rubey
++ Description:
++ This package provides a mapping function for \spadtype{FiniteAbelianMonoidRing}
FiniteAbelianMonoidRingFunctions2(E: OrderedAbelianMonoid,
                                   R1: Ring,
                                   A1: FiniteAbelianMonoidRing(R1, E),
                                   R2: Ring,
                                   A2: FiniteAbelianMonoidRing(R2, E)) _
: Exports == Implementation where

Exports == with

map: (R1 -> R2, A1) -> A2
++ \spad{map}(f, a) applies the map f to each coefficient in a. It is
```

```
++ assumed that f maps 0 to 0
```

```
Implementation == add
```

```
map(f: R1 -> R2, a: A1): A2 ==  
  if zero? a then 0$A2  
  else  
    monomial(f leadingCoefficient a, degree a)$A2 + map(f, reductum a)
```

## 2 package FFFG FractionFreeFastGaussian

```
<package FFFG FractionFreeFastGaussian>≡
)abbrev package FFFG FractionFreeFastGaussian
++ Author: Martin Rubey
++ Description:
++ This package implements the interpolation algorithm proposed in Beckermann,
++ Bernhard and Labahn, George, Fraction-free computation of matrix rational
++ interpolants and matrix GCDs, SIAM Journal on Matrix Analysis and
++ Applications 22.
FractionFreeFastGaussian(D, V): Exports == Implementation where
  D: Join(IntegralDomain, GcdDomain)
  V: AbelianMonoidRing(D, NonNegativeInteger)

SUP ==> SparseUnivariatePolynomial

cFunction ==> (NonNegativeInteger, Vector SUP D) -> D

CoeffAction ==> (NonNegativeInteger, NonNegativeInteger, V) -> D

Exports == with

fffg: (List D, cFunction, List NonNegativeInteger) -> Matrix SUP D
  ++ \spad{fffg} is the general algorithm as proposed by Beckermann and
  ++ Labahn.
  ++
  ++ The second argument, c(k, M) computes c_k(M) as in Equation (2). Note
  ++ that the information about f is therefore encoded in c.

interpolate: (List D, List D, NonNegativeInteger) -> Fraction SUP D
  ++ \spad{interpolate(xlist, ylist, deg)} returns the rational function with
  ++ numerator degree at most \spad{deg} and denominator degree at most
  ++ \spad{#xlist-deg-1} that interpolates the given points using
  ++ fraction free arithmetic. Note that rational interpolation does not
  ++ guarantee that all given points are interpolated correctly:
  ++ unattainable points may make this impossible.
```

**ToDo 2.1.** *The following function could be moved to FFFGF, parallel to generalInterpolation. However, the reason for moving generalInterpolation for fractions to a separate package was the need of a generic signature, hence the extra argument VF to FFFGF. In the special case of rational interpolation, this extra argument is not necessary, since we are always returning a fraction of SUPs, and ignore V. In fact, V is not needed for fffg itself, only if we want to specify a CoeffAction.*

*Thus, maybe it would be better to move fffg to a separate package?*

```

(package FFFG FractionFreeFastGaussian)+≡
  interpolate: (List Fraction D, List Fraction D, NonNegativeInteger)
    -> Fraction SUP D
    ++ \spad{interpolate(xlist, ylist, deg)} returns the rational function with
    ++ numerator degree \spad{deg} that interpolates the given points using
    ++ fraction free arithmetic.

  generalInterpolation: (List D, CoeffAction,
    Vector V, List NonNegativeInteger) -> Matrix SUP D
    ++ \spad{generalInterpolation(C, CA, f, eta)} performs Hermite-Pade
    ++ approximation using the given action CA of polynomials on the elements
    ++ of f. The result is guaranteed to be correct up to order
    ++ |eta|-1. Given that eta is a "normal" point, the degrees on the
    ++ diagonal are given by eta. The degrees of column i are in this case
    ++ eta + e.i - [1,1,...,1], where the degree of zero is -1.
    ++
    ++ The first argument C is the list of coefficients c_{k,k} in the
    ++ expansion <x^k> z g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x).
    ++
    ++ The second argument, CA(k, l, f), should return the coefficient of x^k
    ++ in z^l f(x).

  generalInterpolation: (List D, CoeffAction,
    Vector V, NonNegativeInteger, NonNegativeInteger)
    -> Stream Matrix SUP D
    ++ \spad{generalInterpolation(C, CA, f, sumEta, maxEta)} applies
    ++ \spad{generalInterpolation(C, CA, f, eta)} for all possible \spad{eta}
    ++ with maximal entry \spad{maxEta} and sum of entries at most
    ++ \spad{sumEta}.
    ++
    ++ The first argument C is the list of coefficients c_{k,k} in the
    ++ expansion <x^k> z g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x).
    ++
    ++ The second argument, CA(k, l, f), should return the coefficient of x^k
    ++ in z^l f(x).

  generalCoefficient: (CoeffAction, Vector V,
    NonNegativeInteger, Vector SUP D) -> D

```

```

++ \spad{generalCoefficient(action, f, k, p)} gives the coefficient of
++  $x^k$  in  $p(z)\cdot f(x)$ , where the action of  $z^1$  on a polynomial in  $x$  is
++ given by action, i.e.,  $\text{action}(k, 1, f)$  should return the coefficient
++ of  $x^k$  in  $z^1 f(x)$ .

```

```

ShiftAction: (NonNegativeInteger, NonNegativeInteger, V) -> D
++ \spad{ShiftAction(k, l, g)} gives the coefficient of  $x^k$  in  $z^l g(x)$ ,
++ where  $\text{spad}\{z*(a+b*x+c*x^2+d*x^3+...)\} = (b*x+2*c*x^2+3*d*x^3+...)$ . In
++ terms of sequences,  $z*u(n)=n*u(n)$ .

```

```

ShiftC: NonNegativeInteger -> List D
++ \spad{ShiftC} gives the coefficients  $c_{\{k,k\}}$  in the expansion  $\langle x^k \rangle z$ 
++  $g(x) = \sum_{i=0}^k c_{\{k,i\}} \langle x^i \rangle g(x)$ , where  $z$  acts on  $g(x)$  by
++ shifting. In fact, the result is  $[0,1,2,...]$ 

```

```

DiffAction: (NonNegativeInteger, NonNegativeInteger, V) -> D
++ \spad{DiffAction(k, l, g)} gives the coefficient of  $x^k$  in  $z^l g(x)$ ,
++ where  $z*(a+b*x+c*x^2+d*x^3+...)\} = (a*x+b*x^2+c*x^3+...)$ , i.e.,
++ multiplication with  $x$ .

```

```

DiffC: NonNegativeInteger -> List D
++ \spad{DiffC} gives the coefficients  $c_{\{k,k\}}$  in the expansion  $\langle x^k \rangle z$ 
++  $g(x) = \sum_{i=0}^k c_{\{k,i\}} \langle x^i \rangle g(x)$ , where  $z$  acts on  $g(x)$  by
++ shifting. In fact, the result is  $[0,0,0,...]$ 

```

```

qShiftAction: (D, NonNegativeInteger, NonNegativeInteger, V) -> D
++ \spad{qShiftAction(q, k, l, g)} gives the coefficient of  $x^k$  in  $z^l$ 
++  $g(x)$ , where  $z*(a+b*x+c*x^2+d*x^3+...)\} =$ 
++  $(a+q*b*x+q^2*c*x^2+q^3*d*x^3+...)$ . In terms of sequences,
++  $z*u(n)=q^n*u(n)$ .

```

```

qShiftC: (D, NonNegativeInteger) -> List D
++ \spad{qShiftC} gives the coefficients  $c_{\{k,k\}}$  in the expansion  $\langle x^k \rangle z$ 
++  $g(x) = \sum_{i=0}^k c_{\{k,i\}} \langle x^i \rangle g(x)$ , where  $z$  acts on  $g(x)$  by
++ shifting. In fact, the result is  $[1,q,q^2,...]$ 

```

Implementation ==> add

---

```
-- Shift Operator
```

---

```
-- ShiftAction(k, l, f) is the CoeffAction appropriate for the shift operator.
```

```

ShiftAction(k: NonNegativeInteger, l: NonNegativeInteger, f: V): D ==
  k**l*coefficient(f, k)

```

```

ShiftC(total: NonNegativeInteger): List D ==
  [i::D for i in 0..total-1]

-----
-- q-Shift Operator
-----

-- q-ShiftAction(k, l, f) is the CoeffAction appropriate for the q-shift operator.

qShiftAction(q: D, k: NonNegativeInteger, l: NonNegativeInteger, f: V): D ==
  q**(k*l)*coefficient(f, k)

qShiftC(q: D, total: NonNegativeInteger): List D ==
  [q**i for i in 0..total-1]

-----
-- Differentiation Operator
-----

-- DiffAction(k, l, f) is the CoeffAction appropriate for the differentiation
-- operator.

DiffAction(k: NonNegativeInteger, l: NonNegativeInteger, f: V): D ==
  coefficient(f, (k-1)::NonNegativeInteger)

DiffC(total: NonNegativeInteger): List D ==
  [0 for i in 1..total]

-----
-- general - suitable for functions f
-----

-- get the coefficient of z^k in the scalar product of p and f, the action
-- being defined by coeffAction

generalCoefficient(coeffAction: CoeffAction, f: Vector V,
                  k: NonNegativeInteger, p: Vector SUP D): D ==
  res: D := 0
  for i in 1..#f repeat

-- Defining a and b and summing only over those coefficients that might be
-- nonzero makes a huge difference in speed

```

```
a := f.i
b := p.i
for l in minimumDegree b..degree b repeat
  if not zero? coefficient(b, l)
  then res := res + coefficient(b, l) * coeffAction(k, l, a)
res
```

```
generalInterpolation(C: List D, coeffAction: CoeffAction,
  f: Vector V,
  eta: List NonNegativeInteger): Matrix SUP D ==
c: cFunction := generalCoefficient(coeffAction, f,
  (#1-1)::NonNegativeInteger, #2)
fffg(C, c, eta)
```

```
-----
-- general - suitable for functions f - trying all possible degree combinations
-----
```

The following function returns the lexicographically next vector with non-negative components smaller than  $p$  with the same sum as  $v$ .

```

⟨package FFFG FractionFreeFastGaussian⟩+≡
  nextVector!(p: NonNegativeInteger, v: List NonNegativeInteger)
    : Union("failed", List NonNegativeInteger) ==
    n := #v
    pos := position(#1 < p, v)
    zero? pos => return "failed"
    if pos = 1 then
      sum: Integer := v.1
      for i in 2..n repeat
        if v.i < p and sum > 0 then
          v.i := v.i + 1
          sum := sum - 1
          for j in 1..i-1 repeat
            if sum > p then
              v.j := p
              sum := sum - p
            else
              v.j := sum::NonNegativeInteger
              sum := 0
          return v
        else sum := sum + v.i
      return "failed"
    else
      v.pos := v.pos + 1
      v.(pos-1) := (v.(pos-1) - 1)::NonNegativeInteger

  v

```

The following function returns the stream of all possible degree vectors, beginning with  $v$ , where the degree vectors are sorted in reverse lexicographic order. Furthermore, the entries are all less or equal to  $p$  and their sum equals the sum of the entries of  $v$ . We assume that the entries of  $v$  are also all less or equal to  $p$ .

```

⟨package FFFG FractionFreeFastGaussian⟩+≡
  vectorStream(p: NonNegativeInteger, v: List NonNegativeInteger)
    : Stream List NonNegativeInteger == delay
  next := nextVector!(p, copy v)
  (next case "failed") => empty()$Stream(List NonNegativeInteger)
  cons(next, vectorStream(p, next))

```

`vectorStream2` skips every second entry of `vectorStream`.

```

⟨package FFFG FractionFreeFastGaussian⟩+≡
  vectorStream2(p: NonNegativeInteger, v: List NonNegativeInteger)
    : Stream List NonNegativeInteger == delay
    next := nextVector!(p, copy v)
    (next case "failed") => empty()$Stream(List NonNegativeInteger)
    next2 := nextVector!(p, copy next)
    (next2 case "failed") => cons(next, empty())
    cons(next2, vectorStream2(p, next2))

```

This version of `generalInterpolation` returns a stream of solutions, one for each possible degree vector. Thus, it only needs to apply the previously defined `generalInterpolation` to each degree vector. These are generated by `vectorStream` and `vectorStream2` respectively.

If `f` consists of two elements only, we can skip every second degree vector: note that `fffg`, and thus also `generalInterpolation`, returns a matrix with `#f` columns, each corresponding to a solution of the interpolation problem. More precisely, the  $i^{\text{th}}$  column is a solution with degrees  $\mathbf{eta} - (1, 1, \dots, 1) + e_i$ . Thus, in the case of  $2 \times 2$  matrices, `vectorStream` would produce solutions corresponding to  $(d, 0), (d-1, 1); (d-1, 1), (d-2, 2); (d-2, 2), (d-3, 3) \dots$ , i.e., every second matrix is redundant.

Although some redundancy exists also for higher dimensional `f`, the scheme becomes much more complicated, thus we did not implement it.

```

⟨package FFFG FractionFreeFastGaussian⟩+≡
  generalInterpolation(C: List D, coeffAction: CoeffAction,
    f: Vector V,
    sumEta: NonNegativeInteger,
    maxEta: NonNegativeInteger)
    : Stream Matrix SUP D ==

```

*⟨generate an initial degree vector⟩*

```

if #f = 2 then
  map(generalInterpolation(C, coeffAction, f, #1),
    cons(eta, vectorStream2(maxEta, eta)))
  $StreamFunctions2(List NonNegativeInteger,
    Matrix SUP D)
else
  map(generalInterpolation(C, coeffAction, f, #1),
    cons(eta, vectorStream(maxEta, eta)))
  $StreamFunctions2(List NonNegativeInteger,
    Matrix SUP D)

```

We need to generate an initial degree vector, being the minimal element in reverse lexicographic order, i.e.,  $m, m, \dots, m, k, 0, 0, \dots$ , where  $m$  is `maxEta` and  $k$  is the remainder of `sumEta` divided by `maxEta`. This is done by the following code:

```

<generate an initial degree vector>≡
  sum: Integer := sumEta
  entry: Integer
  eta: List NonNegativeInteger
    := [(if sum < maxEta _
          then (entry := sum; sum := 0) _
          else (entry := maxEta; sum := sum - maxEta); _
         entry::NonNegativeInteger) for i in 1..#f]

```

We want to generate all vectors with sum of entries being at most `sumEta`. Therefore the following is incorrect.

```

<BUG generate an initial degree vector>≡
  -- (sum > 0) => empty()$Stream(Matrix SUP D)

```

*<package FFFG FractionFreeFastGaussian>+≡*

-----  
-- rational interpolation  
-----

```
interpolate(x: List Fraction D, y: List Fraction D, d: NonNegativeInteger)
: Fraction SUP D ==
gx := splitDenominator(x)$InnerCommonDenominator(D, Fraction D, _
List D, _
List Fraction D)
gy := splitDenominator(y)$InnerCommonDenominator(D, Fraction D, _
List D, _
List Fraction D)
r := interpolate(gx.num, gy.num, d)
elt(numer r, monomial(gx.den,1))/(gy.den*elt(denom r, monomial(gx.den,1)))
```

```
interpolate(x: List D, y: List D, d: NonNegativeInteger): Fraction SUP D ==
-- berechne Interpolante mit Graden d und N-d-1
if (N := #x) ~= #y then
error "interpolate: number of points and values must match"
if N <= d then
error "interpolate: numerator degree must be smaller than number of data points"
c: cFunction := y.#1 * elt(#2.2, x.#1) - elt(#2.1, x.#1)
eta: List NonNegativeInteger := [d, (N-d)::NonNegativeInteger]
M := fffg(x, c, eta)

if zero?(M.(2,1)) then M.(1,2)/M.(2,2)
else M.(1,1)/M.(2,1)
```

Because of Lemma 5.3, M.1.(2,1) and M.1.(2,2) cannot both vanish, since  $\eta_{\sigma}$  is always  $\sigma$ -normal by Theorem 7.2 and therefore also para-normal, see Definition 4.2.

Because of Lemma 5.1 we have that M.1.(\*,2) is a solution of the interpolation problem, if M.1.(2,1) vanishes.

`<package FFFG FractionFreeFastGaussian>+≡`

```
-----
-- fffg
-----
```

```
-- a major part of the time is spent here
recurrence(M: Matrix SUP D, lambda: NonNegativeInteger, m: NonNegativeInteger,
           r: Vector D, d: D, z: SUP D, Ck: D, p: Vector D): Matrix SUP D ==
```

```
    rLambda := qelt(r, lambda)
    polyf := rLambda * (z - Ck::SUP D)
```

```
    for i in 1..m repeat
        Milambda := qelt(M, i, lambda)
        newMilambda := polyf * Milambda
```

```
-- update columns  $\tilde{=} \lambda$  and calculate their sum
    for l in 1..m | l  $\tilde{=} \lambda$  repeat
        rl := qelt(r, l)
        Mil := ((qelt(M, i, l) * rLambda - Milambda * rl) exquo d)::SUP D
        qsetelt!(M, i, l, Mil)

        pl := qelt(p, l)
        newMilambda := newMilambda - pl * Mil
```

```
-- update column lambda
```

```
        qsetelt!(M, i, lambda, (newMilambda exquo d)::SUP D)
```

```
    M
```

```
fffg(C: List D, c: cFunction, eta: List NonNegativeInteger): Matrix SUP D ==
```

```
-- eta is the vector of degrees. We compute M with degrees  $\eta + e_{i-1}$ ,  $i=1..m$ 
    z: SUP D := monomial(1, 1)
    m: NonNegativeInteger := #eta
    M: Matrix SUP D := scalarMatrix(m, 1)
    d: D := 1
```

```

K: NonNegativeInteger := reduce(+, eta)
etak: Vector NonNegativeInteger := zero(m)
r: Vector D := zero(m)
p: Vector D := zero(m)
Lambda: List Integer
lambdaMax: Integer
lambda: NonNegativeInteger

    for k in 1..K repeat
-- k = sigma+1

        for l in 1..m repeat r.l := c(k, column(M, l))

        Lambda := [eta.l-etak.l for l in 1..m | r.l ~= 0]

-- if Lambda is empty, then M, d and etak remain unchanged. Otherwise, we look
-- for the next closest para-normal point.

        (empty? Lambda) => "iterate"

        lambdaMax := reduce(max, Lambda)
        lambda := 1
        while eta.lambda-etak.lambda < lambdaMax or r.lambda = 0 repeat
            lambda := lambda + 1

-- Calculate leading coefficients

        for l in 1..m | l ~= lambda repeat
            if etak.l > 0 then
                p.l := coefficient(M.(l, lambda), (etak.l-1)::NonNegativeInteger)
            else
                p.l := 0

-- increase order and adjust degree constraints

        M := recurrence(M, lambda, m, r, d, z, C.k, p)

        d := r.lambda
        etak.lambda := etak.lambda + 1

M

```

### 3 package FFFGF FractionFreeFastGaussianFractions

```
(package FFFGF FractionFreeFastGaussianFractions)≡
)abbrev package FFFGF FractionFreeFastGaussianFractions
++ Author: Martin Rubey
++ Description:
++ This package lifts the interpolation functions from
++ \spadtype{FractionFreeFastGaussian} to fractions.
FractionFreeFastGaussianFractions(D, V, VF): Exports == Implementation where
  D: Join(IntegralDomain, GcdDomain)
  V: FiniteAbelianMonoidRing(D, NonNegativeInteger)
  VF: FiniteAbelianMonoidRing(Fraction D, NonNegativeInteger)

F ==> Fraction D

SUP ==> SparseUnivariatePolynomial

FFFG ==> FractionFreeFastGaussian

FAMR2 ==> FiniteAbelianMonoidRingFunctions2

cFunction ==> (NonNegativeInteger, Vector SUP D) -> D

CoeffAction ==> (NonNegativeInteger, NonNegativeInteger, V) -> D
-- coeffAction(k, l, f) is the coefficient of x^k in z^l f(x)

Exports == with

  generalInterpolation: (List D, CoeffAction, Vector VF, List NonNegativeInteger)
    -> Matrix SUP D
    ++ \spad{generalInterpolation(l, CA, f, eta)} performs Hermite-Pade
    ++ approximation using the given action CA of polynomials on the elements
    ++ of f. The result is guaranteed to be correct up to order
    ++ |eta|-1. Given that eta is a "normal" point, the degrees on the
    ++ diagonal are given by eta. The degrees of column i are in this case
    ++ eta + e.i - [1,1,...,1], where the degree of zero is -1.

  generalInterpolation: (List D, CoeffAction,
    Vector VF, NonNegativeInteger, NonNegativeInteger)
    -> Stream Matrix SUP D
    ++ \spad{generalInterpolation(l, CA, f, sumEta, maxEta)} applies
    ++ generalInterpolation(l, CA, f, eta) for all possible eta with maximal
    ++ entry maxEta and sum of entries sumEta
```

Implementation == add

```
multiplyRows!(v: Vector D, M: Matrix SUP D): Matrix SUP D ==  
  n := #v  
  for i in 1..n repeat  
    for j in 1..n repeat  
      M.(i,j) := v.i*M.(i,j)
```

M

```
generalInterpolation(C: List D, coeffAction: CoeffAction,  
  f: Vector VF, eta: List NonNegativeInteger): Matrix SUP D ==  
  n := #f  
  g: Vector V := new(n, 0)  
  den: Vector D := new(n, 0)
```

```
  for i in 1..n repeat  
    c := coefficients(f.i)  
    den.i := commonDenominator(c)$CommonDenominator(D, F, List F)  
    g.i := map(retract(#1*den.i)@D, f.i)  
            $FAMR2(NonNegativeInteger, Fraction D, VF, D, V)
```

```
  M := generalInterpolation(C, coeffAction, g, eta)$FFFG(D, V)
```

```
-- The following is necessary since I'm multiplying each row with a factor, not  
-- each column. Possibly I could factor out gcd den, but I'm not sure whether  
-- this is efficient.
```

```
  multiplyRows!(den, M)
```

```
generalInterpolation(C: List D, coeffAction: CoeffAction,  
  f: Vector VF,  
  sumEta: NonNegativeInteger,  
  maxEta: NonNegativeInteger)  
: Stream Matrix SUP D ==
```

```
  n := #f  
  g: Vector V := new(n, 0)  
  den: Vector D := new(n, 0)
```

```
  for i in 1..n repeat  
    c := coefficients(f.i)  
    den.i := commonDenominator(c)$CommonDenominator(D, F, List F)  
    g.i := map(retract(#1*den.i)@D, f.i)  
            $FAMR2(NonNegativeInteger, Fraction D, VF, D, V)
```

```

c: cFunction := generalCoefficient(coeffAction, g,
                                   (#1-1)::NonNegativeInteger, #2)$FFFG(D, V)

MS: Stream Matrix SUP D
   := generalInterpolation(C, coeffAction, g, sumEta, maxEta)$FFFG(D, V)

-- The following is necessary since I'm multiplying each row with a factor, not
-- each column. Possibly I could factor out gcd den, but I'm not sure whether
-- this is efficient.

map(multiplyRows!(den, #1), MS)$Stream(Matrix SUP D)

```

## 4 package NEWTON NewtonInterpolation

```
<package NEWTON NewtonInterpolation>≡
)abbrev package NEWTON NewtonInterpolation
++ Description:
++ This package exports Newton interpolation for the special case where the
++ result is known to be in the original integral domain
NewtonInterpolation F: Exports == Implementation where
  F: IntegralDomain
  Exports == with

  newton: List F -> SparseUnivariatePolynomial F

  ++ \spad{newton}(1) returns the interpolating polynomial for the values
  ++ 1, where the x-coordinates are assumed to be [1,2,3,...,n] and the
  ++ coefficients of the interpolating polynomial are known to be in the
  ++ domain F. I.e., it is a very streamlined version for a special case of
  ++ interpolation.

Implementation == add

differences(yl: List F): List F ==
  [y2-y1 for y1 in yl for y2 in rest yl]

z: SparseUnivariatePolynomial(F) := monomial(1,1)

-- we assume x=[1,2,3,...,n]
newtonAux(k: F, fact: F, yl: List F): SparseUnivariatePolynomial(F) ==
  if empty? rest yl
  then ((yl.1) exquo fact)::F::SparseUnivariatePolynomial(F)
  else ((yl.1) exquo fact)::F::SparseUnivariatePolynomial(F)
    + (z-k::SparseUnivariatePolynomial(F)) _
      * newtonAux(k+1$F, fact*k, differences yl)

newton yl == newtonAux(1$F, 1$F, yl)
```

## 5 License

$\langle license \rangle \equiv$

```
-- Copyright (C) 2006 Martin Rubey <Martin.Rubey@univie.ac.at>
--
-- This program is free software; you can redistribute it and/or
-- modify it under the terms of the GNU General Public License as
-- published by the Free Software Foundation; either version 2 of
-- the License, or (at your option) any later version.
--
-- This program is distributed in the hope that it will be
-- useful, but WITHOUT ANY WARRANTY; without even the implied
-- warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
-- PURPOSE. See the GNU General Public License for more details.
```

$\langle * \rangle \equiv$

$\langle license \rangle$

$\langle package\ FAMR2\ FiniteAbelianMonoidRingFunctions2 \rangle$

$\langle package\ FFFG\ FractionFreeFastGaussian \rangle$

$\langle package\ FFFGF\ FractionFreeFastGaussianFractions \rangle$

$\langle package\ NEWTON\ NewtonInterpolation \rangle$