# \$SPAD/src/algebra gaussian.spad

Barry Trager, James Davenport

18 March 1994

**Abstract**

This category represents the extension of a ring by a square root of -1.

`Complex(R)`

creates the domain of elements of the form

`a + b * i`

where

`a`

and b come from the ring R, and i is a new element such that

$\mathtt{i}^2 = -1$

.

# Contents

# 1 category COMPCAT ComplexCategory

⟨*category COMPCAT ComplexCategory*⟩≡

```
)abbrev category COMPCAT ComplexCategory
++ Author:
++ Date Created:
++ Date Last Updated: 18 March 1994
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: complex, gaussian
++ References:
++ Description:
++ This category represents the extension of a ring by a square
++ root of -1.
ComplexCategory(R:CommutativeRing): Category ==
  Join(MonogenicAlgebra(R, SparseUnivariatePolynomial R), FullyRetractableTo R,
   DifferentialExtension R, FullyEvalableOver R, FullyPatternMatchable(R),
    Patternable(R), FullyLinearlyExplicitRingOver R, CommutativeRing) with
     imaginary:   () -> %          ++ imaginary() = sqrt(-1) = %i.
     conjugate:   % -> %           ++ conjugate(x + %i y) returns x - %i y.
     complex  :   (R, R) -> %      ++ complex(x,y) constructs x + %i*y.
     imag     :   % -> R           ++ imag(x) returns imaginary part of x.
     real     :   % -> R           ++ real(x) returns real part of x.
     norm     :   % -> R           ++ norm(x) returns x * conjugate(x)
     if R has Comparable then Comparable
     if R has IntegralDomain then
       IntegralDomain
       _exquo : (%,R) -> Union(%,"failed")
         ++ exquo(x, r) returns the exact quotient of x by r, or
         ++ "failed" if r does not divide x exactly.
     if R has EuclideanDomain then EuclideanDomain
     if R has multiplicativeValuation then multiplicativeValuation
     if R has additiveValuation then additiveValuation
     if R has Field then        -- this is a lie; we must know that
       Field                    -- x^2+1 is irreducible in R
     if R has ConvertibleTo InputForm then ConvertibleTo InputForm
     if R has CharacteristicZero then CharacteristicZero
     if R has CharacteristicNonZero then CharacteristicNonZero
     if R has RealConstant then
       ConvertibleTo Complex DoubleFloat
       ConvertibleTo Complex Float
     if R has RealNumberSystem then
       abs: % -> %
         ++ abs(x) returns the absolute value of x = sqrt(norm(x)).
```

```
    if R has TranscendentalFunctionCategory then
      TranscendentalFunctionCategory
      argument: % -> R  ++ argument(x) returns the angle made by (0,1) and (0,x).
      if R has RadicalCategory then RadicalCategory
      if R has RealNumberSystem then
        polarCoordinates: % -> Record(r:R, phi:R)
          ++ polarCoordinates(x) returns (r, phi) such that x = r * exp(%i * phi).
    if R has IntegerNumberSystem then
      UniqueFactorizationDomain
      rational?    : % -> Boolean
        ++ rational?(x) tests if x is a rational number.
      rational     : % -> Fraction Integer
        ++ rational(x) returns x as a rational number.
        ++ Error: if x is not a rational number.
      rationalIfCan: % -> Union(Fraction Integer, "failed")
        ++ rationalIfCan(x) returns x as a rational number, or
        ++ "failed" if x is not a rational number.
    if R has PolynomialFactorizationExplicit and R has EuclideanDomain then
       PolynomialFactorizationExplicit
add
      import MatrixCategoryFunctions2(%, Vector %, Vector %, Matrix %,
                                      R, Vector R, Vector R, Matrix R)
      SUP ==> SparseUnivariatePolynomial
      characteristicPolynomial x ==
         v := monomial(1,1)$SUP(R)
         v^2 - trace(x)*v^1 + norm(x)*v^0
      if R has PolynomialFactorizationExplicit and R has EuclideanDomain then
         SupR ==> SparseUnivariatePolynomial R
         Sup ==> SparseUnivariatePolynomial %
         import FactoredFunctionUtilities Sup
         import UnivariatePolynomialCategoryFunctions2(R,SupR,%,Sup)
         import UnivariatePolynomialCategoryFunctions2(%,Sup,R,SupR)
         pp,qq:Sup
         if R has IntegerNumberSystem then
            myNextPrime: (%,NonNegativeInteger) -> %
            myNextPrime(x,n ) == -- prime is actually in R, and = 3(mod 4)
               xr:=real(x)-4::R
               while not prime? xr repeat
                  xr:=xr-4::R
               complex(xr,0)
            --!TT:=InnerModularGcd(%,Sup,32719 :: %,myNextPrime)
            --!gcdPolynomial(pp,qq) == modularGcd(pp,qq)$TT
            solveLinearPolynomialEquation(lp:List Sup,p:Sup) ==
              solveLinearPolynomialEquation(lp,p)$ComplexIntegerSolveLinearPolynomialE

         normPolynomial: Sup -> SupR
```

```
   normPolynomial pp ==
       map(retract@(%->R), pp * map(conjugate,pp))
   factorPolynomial pp ==
       refine(squareFree pp,factorSquareFreePolynomial)
   factorSquareFreePolynomial pp ==
       pnorm:=normPolynomial pp
       k:R:=0
       while degree gcd(pnorm,differentiate pnorm)>0 repeat
          k:=k+1
          pnorm:=normPolynomial
                   elt(pp,monomial(1,1)-monomial(complex(0,k),0))
       fR:=factorSquareFreePolynomial pnorm
       numberOfFactors fR = 1 =>
           makeFR(1,[["irred",pp,1]])
       lF:List Record(flg:Union("nil", "sqfr", "irred", "prime"),
                      fctr:Sup, xpnt:Integer):=[]
       for u in factorList fR repeat
           p1:=map((z:R):% +->z::%,u.fctr)
           if not zero? k then
              p1:=elt(p1,monomial(1,1)+monomial(complex(0,k),0))
           p2:=gcd(p1,pp)
           lF:=cons(["irred",p2,1],lF)
           pp:=(pp exquo p2)::Sup
       makeFR(pp,lF)

if R has IntegerNumberSystem then

    factor x ==
        y: Complex Integer := complex(convert(real x)@Integer,
                                      convert(imag x)@Integer)
        z: Factored Complex Integer :=
              factor(y)$GaussianFactorizationPackage
        map(f +-> complex(real(f)::R, imag(f)::R),
            z)$FactoredFunctions2(Complex Integer, %)

    squareFree x == factor x

    prime? x ==
        y: Complex Integer := complex(convert(real x)@Integer,
                                      convert(imag x)@Integer)
        prime?(y)$GaussianFactorizationPackage

rank()           == 2
discriminant()   == -4 :: R
norm x           == real(x)^2 + imag(x)^2
trace x          == 2 * real x
```

```
    imaginary()      == complex(0, 1)
    conjugate x      == complex(real x, - imag x)
    characteristic() == characteristic()$R
    map(fn, x)       == complex(fn real x, fn imag x)
    x = y            == real(x) = real(y) and imag(x) = imag(y)
    x + y            == complex(real x + real y, imag x + imag y)
    - x              == complex(- real x, - imag x)
    r:R * x:%        == complex(r * real x, r * imag x)
    coordinates(x:%) == [real x, imag x]
    n:Integer * x:%  == complex(n * real x, n * imag x)
    differentiate(x:%, d:R -> R) == complex(d real x, d imag x)

    definingPolynomial() ==
      monomial(1,2)$(SUP R) + monomial(1,0)$(SUP R)

    reduce(pol:SUP R) ==
      part:= (monicDivide(pol,definingPolynomial())).remainder
      complex(coefficient(part,0),coefficient(part,1))

    lift(x) == monomial(real x,0)$(SUP R)+monomial(imag x,1)$(SUP R)

    minimalPolynomial x ==
      zero? imag x =>
        monomial(1, 1)$(SUP R) - monomial(real x, 0)$(SUP R)
      monomial(1, 2)$(SUP R) - monomial(trace x, 1)$(SUP R)
        + monomial(norm x, 0)$(SUP R)

    coordinates(x:%, v:Vector %):Vector(R) ==
      ra := real(a := v(minIndex v))
      rb := real(b := v(maxIndex v))
      (#v ~= 2) or
        ((d := recip(ra * (ib := imag b) - (ia := imag a) * rb))
          case "failed") =>error "coordinates: vector is not a basis"
      rx := real x
      ix := imag x
      [d::R * (rx * ib - ix * rb), d::R * (ra * ix - ia * rx)]

    coerce(x:%):OutputForm ==
      re := (r := real x)::OutputForm
      ie := (i := imag x)::OutputForm
      zero? i => re
      outi := '%i::OutputForm
      ip :=
--          one? i => outi
        (i = 1) => outi
--          one?(-i) => -outi
```

```
    ((-i) = 1) => -outi
    ie * outi
  zero? r => ip
  re + ip

retract(x:%):R ==
  not zero?(imag x) =>
    error "Imaginary part is nonzero. Cannot retract."
  real x

retractIfCan(x:%):Union(R, "failed") ==
  not zero?(imag x) => "failed"
  real x

x:% * y:% ==
  complex(real x * real y - imag x * imag y,
           imag x * real y + imag y * real x)

reducedSystem(m:Matrix %):Matrix R ==
  vertConcat(map(real, m), map(imag, m))

reducedSystem(m:Matrix %, v:Vector %):
 Record(mat:Matrix R, vec:Vector R) ==
  rh := reducedSystem(v::Matrix %)@Matrix(R)
  [reducedSystem(m)@Matrix(R), column(rh, minColIndex rh)]

if R has RealNumberSystem then
  abs(x:%):%        == (sqrt norm x)::%

if R has RealConstant then
  convert(x:%):Complex(DoubleFloat) ==
   complex(convert(real x)@DoubleFloat,convert(imag x)@DoubleFloat)

  convert(x:%):Complex(Float) ==
    complex(convert(real x)@Float, convert(imag x)@Float)

if R has ConvertibleTo InputForm then
  convert(x:%):InputForm ==
    convert([convert('complex), convert real x,
             convert imag x]$List(InputForm))@InputForm

if R has ConvertibleTo Pattern Integer then
  convert(x:%):Pattern Integer ==
     convert(x)$ComplexPattern(Integer, R, %)
if R has ConvertibleTo Pattern Float then
  convert(x:%):Pattern Float ==
```

```
        convert(x)$ComplexPattern(Float, R, %)

    if R has PatternMatchable Integer then
      patternMatch(x:%, p:Pattern Integer,
       l:PatternMatchResult(Integer, %)) ==
        patternMatch(x, p, l)$ComplexPatternMatch(Integer, R, %)

    if R has PatternMatchable Float then
      patternMatch(x:%, p:Pattern Float,
       l:PatternMatchResult(Float, %)) ==
        patternMatch(x, p, l)$ComplexPatternMatch(Float, R, %)

    if R has Comparable then
      smaller?(x, y) ==
        real x = real y => smaller?(imag x, imag y)
        smaller?(real x, real y)

    if R has IntegerNumberSystem then
      rational? x == zero? imag x

      rational x ==
        zero? imag x => rational real x
        error "Not a rational number"

      rationalIfCan x ==
        zero? imag x => rational real x
        "failed"

    if R has Field then
      inv x ==
        zero? imag x => (inv real x)::%
        r := norm x
        complex(real(x) / r, - imag(x) / r)

    if R has IntegralDomain then
      _exquo(x:%, r:R) ==
--          one? r => x
        (r = 1) => x
        (r1 := real(x) exquo r) case "failed" => "failed"
        (r2 := imag(x) exquo r) case "failed" => "failed"
        complex(r1, r2)

      _exquo(x:%, y:%) ==
        zero? imag y => x exquo real y
        x * conjugate(y) exquo norm(y)
```

```
  recip(x:%) == 1 exquo x

  if R has OrderedRing then
    unitNormal x ==
      zero? x => [1,x,1]
      (u := recip x) case % => [x, 1, u]
      zero? real x =>
        c := unitNormal imag x
        [complex(0, c.unit), (c.associate * imag x)::%,
                                       complex(0, - c.associate)]
      c := unitNormal real x
      x := c.associate * x
      imag x < 0 =>
        x := complex(- imag x, real x)
        [- c.unit * imaginary(), x, c.associate * imaginary()]
      [c.unit ::%, x, c.associate ::%]
  else
    unitNormal x ==
      zero? x => [1,x,1]
      (u := recip x) case % => [x, 1, u]
      zero? real x =>
        c := unitNormal imag x
        [complex(0, c.unit), (c.associate * imag x)::%,
                                       complex(0, - c.associate)]
      c := unitNormal real x
      x := c.associate * x
      [c.unit ::%, x, c.associate ::%]

if R has EuclideanDomain then
   if R has additiveValuation then
       euclideanSize x == max(euclideanSize real x,
                              euclideanSize imag x)
   else
       euclideanSize x == euclideanSize(real(x)^2 + imag(x)^2)$R
   if R has IntegerNumberSystem then
     x rem y ==
       zero? imag y =>
         yr:=real y
         complex(symmetricRemainder(real(x), yr),
                 symmetricRemainder(imag(x), yr))
       divide(x, y).remainder
     x quo y ==
       zero? imag y =>
         yr:= real y
         xr:= real x
         xi:= imag x
```

```
      complex((xr-symmetricRemainder(xr,yr)) quo yr,
              (xi-symmetricRemainder(xi,yr)) quo yr)
    divide(x, y).quotient

  else
    x rem y ==
      zero? imag y =>
        yr:=real y
        complex(real(x) rem yr,imag(x) rem yr)
      divide(x, y).remainder
    x quo y ==
      zero? imag y => complex(real x quo real y,imag x quo real y)
      divide(x, y).quotient

  divide(x, y) ==
    r := norm y
    y1 := conjugate y
    xx := x * y1
    x1 := real(xx) rem r
    a  := x1
    if x1 ~= 0 and sizeLess?(r, 2 * x1) then
      a := x1 - r
      if sizeLess?(x1, a) then a := x1 + r
    x2 := imag(xx) rem r
    b  := x2
    if x2 ~= 0 and sizeLess?(r, 2 * x2) then
      b := x2 - r
      if sizeLess?(x2, b) then b := x2 + r
    y1 := (complex(a, b) exquo y1)::%
    [((x - y1) exquo y)::%, y1]

if R has TranscendentalFunctionCategory then
  half := recip(2::R)::R

  if R has RealNumberSystem then
    atan2loc(y: R, x: R): R ==
        pi1 := pi()$R
        pi2 := pi1 * half
        x = 0 => if y >= 0 then pi2 else -pi2

        -- Atan in (-pi/2,pi/2]
        theta := atan(y * recip(x)::R)
        while theta <= -pi2 repeat theta := theta + pi1
        while theta >   pi2 repeat theta := theta - pi1

        x >= 0 => theta      -- I or IV
```

```
      if y >= 0 then
          theta + pi1      -- II
      else
          theta - pi1      -- III

  argument x == atan2loc(imag x, real x)

else
  if R has RadicalCategory then
    argument x ==
      n1 := sqrt(norm(x))
      x1 := real(x) + n1
      (2::R)*atan(imag(x) * recip(x1)::R)

  else
    -- Emulate sqrt using exp and log
    argument x ==
      n1 := exp(half*log(norm(x)))
      x1 := real(x) + n1
      (2::R)*atan(imag(x) * recip(x1)::R)

pi()  == pi()$R :: %

if R is DoubleFloat then
   stoc ==> S_-TO_-C$Lisp
   ctos ==> C_-TO_-S$Lisp

   exp   x == ctos EXP(stoc x)$Lisp
   log   x == ctos LOG(stoc x)$Lisp

   sin   x == ctos SIN(stoc x)$Lisp
   cos   x == ctos COS(stoc x)$Lisp
   tan   x == ctos TAN(stoc x)$Lisp
   asin  x == ctos ASIN(stoc x)$Lisp
   acos  x == ctos ACOS(stoc x)$Lisp
   atan  x == ctos ATAN(stoc x)$Lisp

   sinh  x == ctos SINH(stoc x)$Lisp
   cosh  x == ctos COSH(stoc x)$Lisp
   tanh  x == ctos TANH(stoc x)$Lisp
   asinh x == ctos ASINH(stoc x)$Lisp
   acosh x == ctos ACOSH(stoc x)$Lisp
   atanh x == ctos ATANH(stoc x)$Lisp

else
```

```
  atan x ==
    ix := imaginary()*x
    - imaginary() * half * (log(1 + ix) - log(1 - ix))

  log x ==
    complex(log(norm x) * half, argument x)

  exp x ==
    e := exp real x
    complex(e * cos imag x, e * sin imag x)

  cos x ==
    e := exp(imaginary() * x)
    half * (e + recip(e)::%)

  sin x ==
    e := exp(imaginary() * x)
    - imaginary() * half * (e - recip(e)::%)

if R has RealNumberSystem then
  polarCoordinates x ==
    [sqrt norm x, (negative?(t := argument x) => t + 2 * pi(); t)]

  x:% ^ q:Fraction(Integer) ==
    zero? q =>
      zero? x => error "0 ^ 0 is undefined"
      1
    zero? x => 0
    rx := real x
    zero? imag x and positive? rx => (rx ^ q)::%
    zero? imag x and denom q = 2 => complex(0, (-rx)^q)
    ax := sqrt(norm x) ^ q
    tx := q::R * argument x
    complex(ax * cos tx, ax * sin tx)

else if R has RadicalCategory then
  x:% ^ q:Fraction(Integer) ==
    zero? q =>
      zero? x => error "0 ^ 0 is undefined"
      1
    r := real x
    zero?(i := imag x) => (r ^ q)::%
    t := numer(q) * recip(denom(q)::R)::R * argument x
    e:R :=
      zero? r => i ^ q
      norm(x) ^ (q / (2::Fraction(Integer)))
```

```
        complex(e * cos t, e * sin t)
```

# 2 package COMPLPAT ComplexPattern

⟨*package COMPLPAT ComplexPattern*⟩≡

```
)abbrev package COMPLPAT ComplexPattern
++ Author: Barry Trager
++ Date Created: 30 Nov 1995
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: complex, patterns
++ References:
++ Description:
++ This package supports converting complex expressions to patterns
ComplexPattern(R, S, CS) : C == T where
    R: SetCategory
    S: Join(ConvertibleTo Pattern R, CommutativeRing)
    CS: ComplexCategory S
    C == with
       convert: CS -> Pattern R
          ++ convert(cs) converts the complex expression cs to a pattern

    T == add

       ipat : Pattern R := patternVariable('%i, true, false, false)

       convert(cs) ==
          zero? imag cs => convert real cs
          convert real cs + ipat * convert imag cs
```

# 3 package CPMATCH ComplexPatternMatch

⟨*package CPMATCH ComplexPatternMatch*⟩≡

```
)abbrev package CPMATCH ComplexPatternMatch
++ Author: Barry Trager
++ Date Created: 30 Nov 1995
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: complex, pattern matching
++ References:
++ Description:
++ This package supports matching patterns involving complex expressions
ComplexPatternMatch(R, S, CS) : C == T where
    R: SetCategory
    S: Join(PatternMatchable R, CommutativeRing)
    CS: ComplexCategory S
    PMRS ==> PatternMatchResult(R, CS)
    PS   ==> Polynomial S
    C == with
       if PS has PatternMatchable(R) then
           patternMatch: (CS, Pattern R, PMRS) -> PMRS
             ++ patternMatch(cexpr, pat, res) matches the pattern pat to the
             ++ complex expression cexpr. res contains the variables of pat
             ++ which are already matched and their matches.

    T == add

       import PatternMatchPushDown(R, S, CS)
       import PatternMatchResultFunctions2(R, PS, CS)
       import PatternMatchResultFunctions2(R, CS, PS)

       ivar : PS := '%i::PS

       makeComplex(p:PS):CS ==
          up := univariate p
          degree up > 1 => error "not linear in %i"
          icoef:=leadingCoefficient(up)
          rcoef:=leadingCoefficient(reductum p)
          complex(rcoef,icoef)

       makePoly(cs:CS):PS == real(cs)*ivar + imag(cs)::PS

       if PS has PatternMatchable(R) then
```

```
patternMatch(cs, pat, result) ==
   zero? imag cs =>
      patternMatch(real cs, pat, result)
   map(makeComplex,
      patternMatch(makePoly cs, pat, map(makePoly, result)))
```

# 4 domain COMPLEX Complex

⟨*domain COMPLEX Complex*⟩≡

```
)abbrev domain COMPLEX Complex
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ \spadtype {Complex(R)} creates the domain of elements of the form
++ \spad{a + b * i} where \spad{a} and b come from the ring R,
++ and i is a new element such that \spad{i^2 = -1}.
Complex(R:CommutativeRing): ComplexCategory(R) with
     if R has OpenMath then OpenMath
   == add
       Rep := Record(real:R, imag:R)

       if R has OpenMath then
         writeOMComplex(dev: OpenMathDevice, x: %): Void ==
          OMputApp(dev)
          OMputSymbol(dev, "complex1", "complex__cartesian")
          OMwrite(dev, real x)
          OMwrite(dev, imag x)
          OMputEndApp(dev)

         OMwrite(x: %): String ==
          s: String := ""
          sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
          dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML)
          OMputObject(dev)
          writeOMComplex(dev, x)
          OMputEndObject(dev)
          OMclose(dev)
          s := OM_-STRINGPTRTOSTRING(sp)$Lisp pretend String
          s

         OMwrite(x: %, wholeObj: Boolean): String ==
          s: String := ""
          sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
          dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML)
          if wholeObj then
```

```
        OMputObject(dev)
      writeOMComplex(dev, x)
      if wholeObj then
        OMputEndObject(dev)
      OMclose(dev)
      s := OM_-STRINGPTRTOSTRING(sp)$Lisp pretend String
      s

     OMwrite(dev: OpenMathDevice, x: %): Void ==
      OMputObject(dev)
      writeOMComplex(dev, x)
      OMputEndObject(dev)

     OMwrite(dev: OpenMathDevice, x: %, wholeObj: Boolean): Void ==
      if wholeObj then
        OMputObject(dev)
      writeOMComplex(dev, x)
      if wholeObj then
        OMputEndObject(dev)

   0                == [0, 0]
   1                == [1, 0]
   zero? x          == zero?(x.real) and zero?(x.imag)
--       one? x           == one?(x.real) and zero?(x.imag)
   one? x           == ((x.real) = 1) and zero?(x.imag)
   coerce(r:R):%    == [r, 0]
   complex(r, i)   == [r, i]
   real x           == x.real
   imag x           == x.imag
   x + y            == [x.real + y.real, x.imag + y.imag]
                       -- by re-defining this here, we save 5 fn calls
   x:% * y:% ==
     [x.real * y.real - x.imag * y.imag,
      x.imag * y.real + y.imag * x.real] -- here we save nine!


   if R has IntegralDomain then
     _exquo(x:%, y:%) == -- to correct bad defaulting problem
       zero? y.imag => x exquo y.real
       x * conjugate(y) exquo norm(y)
```

# 5 package COMPLEX2 ComplexFunctions2

⟨*package COMPLEX2 ComplexFunctions2*⟩≡

```
)abbrev package COMPLEX2 ComplexFunctions2
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   This package extends maps from underlying rings to maps between
++   complex over those rings.
ComplexFunctions2(R:CommutativeRing, S:CommutativeRing): with
    map:     (R -> S, Complex R) -> Complex S
      ++ map(f,u) maps f onto real and imaginary parts of u.
 == add
    map(fn, gr) == complex(fn real gr, fn imag gr)
```

# 6 package COMPFACT ComplexFactorization

⟨*package COMPFACT ComplexFactorization*⟩≡

```
)abbrev package COMPFACT ComplexFactorization
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors: Complex, UnivariatePolynomial
++ Also See:
++ AMS Classifications:
++ Keywords: complex, polynomial factorization, factor
++ References:
ComplexFactorization(RR,PR) : C == T where
  RR   :    EuclideanDomain   -- R is Z or Q
  PR   :    UnivariatePolynomialCategory Complex RR
  R    ==>  Complex RR
  I    ==>  Integer
  RN   ==>  Fraction I
  GI   ==>  Complex I
  GRN  ==>  Complex RN


  C  == with

     factor        :   PR   ->  Factored PR
       ++ factor(p) factorizes the polynomial p with complex coefficients.

  T  == add
     SUP    ==> SparseUnivariatePolynomial
     fUnion ==> Union("nil", "sqfr", "irred", "prime")
     FF     ==> Record(flg:fUnion, fctr:PR, xpnt:Integer)
     SAEF   :=  SimpleAlgebraicExtensionAlgFactor(SUP RN,GRN,SUP GRN)
     UPCF2  :=  UnivariatePolynomialCategoryFunctions2(R,PR,GRN,SUP GRN)
     UPCFB  :=  UnivariatePolynomialCategoryFunctions2(GRN,SUP GRN,R,PR)

     myMap(r:R) : GRN ==
       R is GI   =>
         cr :GI := r pretend GI
         complex((real cr)::RN,(imag cr)::RN)
       R is GRN  => r pretend GRN

     compND(cc:GRN):Record(cnum:GI,cden:Integer) ==
       ccr:=real cc
       cci:=imag cc
       dccr:=denom ccr
```

```
  dcci:=denom cci
  ccd:=lcm(dccr,dcci)
  [complex(((ccd exquo dccr)::Integer)*numer ccr,
           ((ccd exquo dcci)::Integer)*numer cci),ccd]

conv(f:SUP GRN) :Record(convP:SUP GI, convD:RN) ==
  pris:SUP GI :=0
  dris:Integer:=1
  dris1:Integer:=1
  pdris:Integer:=1
  for i in 0..(degree f) repeat
    (cf:= coefficient(f,i)) = 0 => "next i"
    cdf:=compND cf
    dris:=lcm(cdf.cden,dris1)
    pris:=((dris exquo dris1)::Integer)*pris +
          ((dris exquo cdf.cden)::Integer)*
            monomial(cdf.cnum,i)$(SUP GI)
    dris1:=dris
  [pris,dris::RN]

backConv(ffr:Factored SUP GRN) : Factored PR ==
  R is GRN =>
    makeFR((unit ffr) pretend PR,[[f.flg,(f.fctr) pretend PR,f.xpnt]
                                   for f in factorList ffr])
  R is GI  =>
    const:=unit ffr
    ris: List FF :=[]
    for ff in factorList ffr repeat
      fact:=primitivePart(conv(ff.fctr).convP)
      expf:=ff.xpnt
      ris:=cons([ff.flg,fact pretend PR,expf],ris)
      lc:GRN := myMap leadingCoefficient(fact pretend PR)
      const:= const*(leadingCoefficient(ff.fctr)/lc)^expf
    uconst:GI:= compND(coefficient(const,0)).cnum
    makeFR((uconst pretend R)::PR,ris)


factor(pol : PR)  : Factored PR ==
  ratPol:SUP GRN := 0
  ratPol:=map(myMap,pol)$UPCF2
  ffr:=factor ratPol
  backConv ffr
```

# 7 package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation

⟨*package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation*⟩≡

```
)abbrev package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation
++ Author: James Davenport
++ Date Created: 1990
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package provides the generalized euclidean algorithm which is
++ needed as the basic step for factoring polynomials.
ComplexIntegerSolveLinearPolynomialEquation(R,CR): C == T
 where
  CP ==> SparseUnivariatePolynomial CR
  R:IntegerNumberSystem
  CR:ComplexCategory(R)
  C == with
      solveLinearPolynomialEquation: (List CP,CP) -> Union(List CP,"failed")
                   ++ solveLinearPolynomialEquation([f1, ..., fn], g)
                   ++ where (fi relatively prime to each other)
                   ++ returns a list of ai such that
                   ++ g = sum ai prod fj (j \= i) or
                   ++ equivalently g/prod fj = sum (ai/fi)
                   ++ or returns "failed" if no such list exists
  T == add
      oldlp:List CP := []
      slpePrime:R:=(2::R)
      oldtable:Vector List CP := empty()
      solveLinearPolynomialEquation(lp,p) ==
         if (oldlp ~= lp) then
            -- we have to generate a new table
            deg:= _+/[degree u for u in lp]
            ans:Union(Vector List CP,"failed"):="failed"
            slpePrime:=67108859::R   -- 2^26 -5 : a prime
                 -- a good test case for this package is
                 --  (good question?)
            while (ans case "failed") repeat
              ans:=tablePow(deg,complex(slpePrime,0),lp)$GenExEuclid(CR,CP)
              if (ans case "failed") then
```

```
              slpePrime:=  slpePrime-4::R
              while not prime?(slpePrime)$IntegerPrimesPackage(R) repeat
                slpePrime:= slpePrime-4::R
          oldtable:=(ans:: Vector List CP)
       answer:=solveid(p,complex(slpePrime,0),oldtable)
       answer
```

# 8 License

⟨*license*⟩≡

```
--Copyright (c) 1991-2002, The Numerical ALgorithms Group Ltd.
--All rights reserved.
--
--Redistribution and use in source and binary forms, with or without
--modification, are permitted provided that the following conditions are
--met:
--
--    - Redistributions of source code must retain the above copyright
--      notice, this list of conditions and the following disclaimer.
--
--    - Redistributions in binary form must reproduce the above copyright
--      notice, this list of conditions and the following disclaimer in
--      the documentation and/or other materials provided with the
--      distribution.
--
--    - Neither the name of The Numerical ALgorithms Group Ltd. nor the
--      names of its contributors may be used to endorse or promote products
--      derived from this software without specific prior written permission.
--
--THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
--IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
--TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
--PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
--OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
--EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
--PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
--PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
--LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
--NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
--SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

⟨*⟩≡
⟨*license*⟩

⟨*category COMPCAT ComplexCategory*⟩
⟨*package COMPLPAT ComplexPattern*⟩
⟨*package CPMATCH ComplexPatternMatch*⟩
⟨*domain COMPLEX Complex*⟩
⟨*package COMPLEX2 ComplexFunctions2*⟩
⟨*package COMPFACT ComplexFactorization*⟩
⟨*package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation*⟩

## References

[1] nothing