$SPAD/src/algebra geneez.spad

Patrizia Gianni

December 8, 2007

Abstract

Contents

1 package GENEEZ GenExEuclid

⟨*package GENEEZ GenExEuclid*⟩≡

```
)abbrev package GENEEZ GenExEuclid
++ Author : P.Gianni.
++ January 1990
++ The equation \spad{Af+Bg=h} and its generalization to n polynomials
++ is solved for solutions over the R, euclidean domain.
++ A table containing the solutions of \spad{Af+Bg=x**k} is used.
++ The operations are performed modulus a prime which are in principle big enough,
++ but the solutions are tested and, in case of failure, a hensel
++ lifting process is used to get to the right solutions.
++ It will be used in the factorization of multivariate polynomials
++ over finite field, with \spad{R=F[x]}.

GenExEuclid(R,BP) : C == T
 where
  R   :   EuclideanDomain
  PI  ==> PositiveInteger
  NNI ==> NonNegativeInteger
  BP  :   UnivariatePolynomialCategory R
  L   ==> List

  C == with
       reduction: (BP,R) -> BP
         ++ reduction(p,prime) reduces the polynomial p modulo prime of R.
         ++ Note: this function is exported only because it's conditional.
       compBound: (BP,L BP) -> NNI
         ++ compBound(p,lp)
         ++ computes a bound for the coefficients of the solution
         ++ polynomials.
         ++ Given a polynomial right hand side p, and a list lp of left hand side polynomi
         ++ Exported because it depends on the valuation.
       tablePow :    (NNI,R,L BP)     -> Union(Vector(L BP),"failed")
         ++ tablePow(maxdeg,prime,lpol) constructs the table with the
         ++ coefficients of the Extended Euclidean Algorithm for lpol.
         ++ Here the right side is \spad{x**k}, for k less or equal to maxdeg.
         ++ The operation returns "failed" when the elements are not coprime modulo prime.
       solveid  : (BP,R,Vector L BP) -> Union(L BP,"failed")
         ++ solveid(h,table) computes the coefficients of the
         ++ extended euclidean algorithm for a list of polynomials
         ++ whose tablePow is table and with right side h.

       testModulus : (R, L BP)    -> Boolean
         ++ testModulus(p,lp) returns true if the the prime p
         ++ is valid for the list of polynomials lp, i.e. preserves
```

```
    ++ the degree and they remain relatively prime.

T == add
  if R has multiplicativeValuation then
    compBound(m:BP,listpolys:L BP) : NNI ==
      ldeg:=[degree f for f in listpolys]
      n:NNI:= (+/[df for df in ldeg])
      normlist:=[ +/[euclideanSize(u)**2 for u in coefficients f]
                        for f in listpolys]
      nm:= +/[euclideanSize(u)**2 for u in coefficients m]
      normprod := */[g**((n-df)::NNI) for g in normlist for df in ldeg]
      2*(approxSqrt(normprod * nm)$IntegerRoots(Integer))::NNI
  else if R has additiveValuation then
    -- a fairly crude Hadamard-style bound for the solution
    -- based on regarding the problem as a system of linear equations.
    compBound(m:BP,listpolys:L BP) : NNI ==
      "max"/[euclideanSize u for u in coefficients m] +
        +/["max"/[euclideanSize u for u in coefficients p]
           for p in listpolys]
  else
    compBound(m:BP,listpolys:L BP) : NNI ==
      error "attempt to use compBound without a well-understood valuation"
  if R has IntegerNumberSystem then
    reduction(u:BP,p:R):BP ==
      p = 0 => u
      map(symmetricRemainder(#1,p),u)
  else reduction(u:BP,p:R):BP ==
      p = 0 => u
      map(#1 rem p,u)

  merge(p:R,q:R):Union(R,"failed") ==
       p = q => p
       p = 0 => q
       q = 0 => p
       "failed"

  modInverse(c:R,p:R):R ==
      (extendedEuclidean(c,p,1)::Record(coef1:R,coef2:R)).coef1

  exactquo(u:BP,v:BP,p:R):Union(BP,"failed") ==
      invlcv:=modInverse(leadingCoefficient v,p)
      r:=monicDivide(u,reduction(invlcv*v,p))
      reduction(r.remainder,p) ^=0 => "failed"
      reduction(invlcv*r.quotient,p)

  FP:=EuclideanModularRing(R,BP,R,reduction,merge,exactquo)
```

```
--make table global variable!
table:Vector L BP
import GeneralHenselPackage(R,BP)

                   --local  functions
makeProducts :    L BP   -> L BP
liftSol: (L BP,BP,R,R,Vector L BP,BP,NNI) -> Union(L BP,"failed")

reduceList(lp:L BP,lmod:R): L FP ==[reduce(ff,lmod) for ff in lp]

coerceLFP(lf:L FP):L BP == [fm::BP for fm in lf]

liftSol(oldsol:L BP,err:BP,lmod:R,lmodk:R,
       table:Vector L BP,m:BP,bound:NNI):Union(L BP,"failed") ==
  euclideanSize(lmodk) > bound => "failed"
  d:=degree err
  ftab:Vector L FP :=
    map(reduceList(#1,lmod),table)$VectorFunctions2(List BP,List FP)
  sln:L FP:=[0$FP for xx in ftab.1 ]
  for i in 0 .. d |(cc:=coefficient(err,i)) ^=0 repeat
    sln:=[slp+reduce(cc::BP,lmod)*pp
          for pp in ftab.(i+1) for slp in sln]
  nsol:=[f-lmodk*reduction(g::BP,lmod) for f in oldsol for g in sln]
  lmodk1:=lmod*lmodk
  nsol:=[reduction(slp,lmodk1) for slp in nsol]
  lpolys:L BP:=table.(#table)
  (fs:=+/[f*g for f in lpolys for g in nsol]) = m => nsol
  a:BP:=((fs-m) exquo lmodk1)::BP
  liftSol(nsol,a,lmod,lmodk1,table,m,bound)

makeProducts(listPol:L BP):L BP ==
  #listPol < 2 => listPol
  #listPol = 2 => reverse listPol
  f:= first listPol
  ll := rest listPol
  [*/ll,:[f*g for g in makeProducts ll]]

testModulus(pmod, listPol) ==
     redListPol := reduceList(listPol, pmod)
     for pol in listPol for rpol in redListPol repeat
          degree(pol) ^= degree(rpol::BP) => return false
     while not empty? redListPol repeat
         rpol := first redListPol
         redListPol := rest redListPol
         for rpol2 in redListPol repeat
```

```
            gcd(rpol, rpol2) ^= 1 => return false
     true

if R has Field then
  tablePow(mdeg:NNI,pmod:R,listPol:L BP) ==
    multiE:=multiEuclidean(listPol,1$BP)
    multiE case "failed" => "failed"
    ptable:Vector L BP :=new(mdeg+1,[])
    ptable.1:=multiE
    x:BP:=monomial(1,1)
    for i in 2..mdeg repeat ptable.i:=
        [tpol*x rem fpol for tpol in ptable.(i-1) for fpol in listPol]
    ptable.(mdeg+1):=makeProducts listPol
    ptable

  solveid(m:BP,pmod:R,table:Vector L BP) : Union(L BP,"failed") ==
        -- Actually, there's no possibility of failure
    d:=degree m
    sln:L BP:=[0$BP for xx in table.1]
    for i in 0 .. d | coefficient(m,i)^=0 repeat
      sln:=[slp+coefficient(m,i)*pp
            for pp in table.(i+1) for slp in sln]
    sln

else

  tablePow(mdeg:NNI,pmod:R,listPol:L BP) ==
    listP:L FP:= [reduce(pol,pmod) for pol in listPol]
    multiE:=multiEuclidean(listP,1$FP)
    multiE case "failed" => "failed"
    ftable:Vector L FP :=new(mdeg+1,[])
    fl:L FP:= [ff::FP for ff in multiE]
    ftable.1:=[fpol for fpol in fl]
    x:FP:=reduce(monomial(1,1),pmod)
    for i in 2..mdeg repeat ftable.i:=
        [tpol*x rem fpol for tpol in ftable.(i-1) for fpol in listP]
    ptable:= map(coerceLFP,ftable)$VectorFunctions2(List FP,List BP)
    ptable.(mdeg+1):=makeProducts listPol
    ptable

  solveid(m:BP,pmod:R,table:Vector L BP) : Union(L BP,"failed") ==
    d:=degree m
    ftab:Vector L FP:=
      map(reduceList(#1,pmod),table)$VectorFunctions2(List BP,List FP)
    lpolys:L BP:=table.(#table)
    sln:L FP:=[0$FP for xx in ftab.1]
```

```
for i in 0 .. d | coefficient(m,i)^=0 repeat
  sln:=[slp+reduce(coefficient(m,i)::BP,pmod)*pp
        for pp in ftab.(i+1) for slp in sln]
soln:=[slp::BP for slp in sln]
(fs:=+/[f*g for f in lpolys for g in soln]) = m=> soln
-- Compute bound
bound:=compBound(m,lpolys)
a:BP:=((fs-m) exquo pmod)::BP
liftSol(soln,a,pmod,pmod,table,m,bound)
```

2 License

⟨*license*⟩≡

$\langle\, *\rangle\equiv$
$\langle$*license*$\rangle$

$\langle$*package GENEEZ GenExEuclid*$\rangle$

References

[1] nothing