```
/**********************************************************************
Date: Mon, 29 Jan 2001 01:26:54 -0500 (EST)
From: Robert Pollack <rpollack@math.harvard.edu>
To: "William A. Stein" <was@math.harvard.edu>
Subject: Lseries code

> Hey, send me some magma code that involves p-adic L-functions!

Okay...but the code is really awful.  I believe it works though.

I will send you two more e-mails.  One containing a write-up in
TeX of computing the p-adic L-series and the other will be my
code.  The first will probably be more helpful than the later.

As the code is not documented, let me at least lead you thru
what it is supposed to do.  The main procedure is called Lseries.
It takes in five parameters - let's call them V,p,a_p,n,D.

The V should be the space of modular symbols associated to E
The p is the prime
The a_p is the a_p value of E
The n represents the level of accuracy.
The D represents twisting by a quadratic character of conductor D.
         (you can always just take D=1)

Then Lseries(V,p,a_p,n,D) returns in the ordinary case a polynomial
of degree p^(n-1)-1 approximating the L-series.  In the supersingular
case, it returns two polynomials say G and H so that G + H * alpha
approximates the L-series where alpha is a root of x^2 - a_p x + p.

The other main procedure is iwasawa_invariants_w_curve.  This
takes 6 arguments, say N,V,p,a_p,n,D.

Here N is the conductor of the curve
     V is the space of modular symbols of the curve
     p is some prime
     a_p is the a_p value of the curve
     n is the level of accuracy
     D represents twisting by a quadratic character of conductor D

This procedure is supposed to return the iwasawa invariants of
the L-series.

Precisely, in the ordinary case it returns a 6-tuple of data,
namely the rank of the curve,
       the p-adic valuation of the leading non-zero term of the L-series,
       the mu invariant of the L-series
       the lambda invariant of the L-series
       the newton polygon of the L-series.
                -this is returned as [ [m_1,s_1] , [m_2,s_2] , ... ]
```

```
                 where m_1 is the number of roots of slope s_1...;
                 a slope of 10000 represents the root 0.
       a boolean saying whether the above data is correct.
         (note if mu is positive then a "true" only means that the data
         is as accurate as it can be using a polynomial of degree
         p^(n-1)-1; that is it could have a another root of smaller slope)


In the supersingular case, the p-adic L-series is not an iwasawa
function so the above data couldn't make any sense.  Instead it
returns an 11-tuple of data consisting of the rank and then the
same data for the g and h functions i construct in my thesis.  i should
have a rough draft of my thesis ready soon if you are interested
in how they are defined.  basically, they are the "real" and "imaginary"
parts of the L-series with their trivial zeroes removed.  They are
in fact iwasawa functions (actually integral power series).

I've been back and forth between NY and Boston for the last
few weeks.  Now I should be back in Boston full time for the
next month.

Okay...let me try and send out the code.

Robert
*******************************************************************/

R<x>:=PolynomialRing(Rationals());
R<T>:=PolynomialRing(Rationals());

//This function cuts out the subspace of H_1(X_0(N),Z) where
//T_2 acts as a_2, T_3 acts as a_3, T_5 acts as a_5 and T_7 acts as a_7
//Input:   N - conductor
// a_2 - eigenvalue of T_2
// a_3 - eigenvalue of T_3
// a_5 - eigenvalue of T_5
// a_7 - eigenvalue of T_7
//
//Output: space of modular symbols of level N and specified eigenvalues
function form_curve(N,a_2,a_3,a_5,a_7)

R<x>:=PolynomialRing(Rationals());
M:=ModularSymbols(N,2,+1);
I:=[<2,x-a_2>,<3,x-a_3>,<5,x-a_5>,<7,x-a_7>];
V:=Kernel(I,M);
ModularSymbolEven(V,1/2);

return(V);
end function;

function table_of_modular_symbols(V,p,n,D)
```

```
local q,table,temp,k,scale,q1,mid,temp2,e;

if (D mod p eq 0) then
print "Twisting by something not relatively prime to p!";
end if;

if (KroneckerSymbol(D,-1) eq -1) then
print "Twisting by an odd character!";
end if;

q1 := p^(n-1);
q := q1*p;

k:=1;
while (ModularSymbolEven(V,k/p^10)[1] eq 0) do
k := k+1;
end while;

scale := Denominator(ModularSymbolEven(V,k/q)[1]);
table := [];

for a in [0..(q-1) div 2] do
if (Valuation(a,p) le 2) or (a eq 0) then
temp2 :=0;
for b in [1..D] do
e := KroneckerSymbol(D,b);
if (e ne 0) then
temp := ModularSymbolEven(V,(D*a-q*b)/(D*q))[1] * scale;
if Denominator(temp) ne 1 then
print "PROBLEMS!  There are still denominators in the modsyms.";
end if;
temp2 := temp2 + KroneckerSymbol(D,b)*temp;
end if;
end for;
table := table cat [Numerator(temp2)];
else
table := table cat [0];
end if;
end for;

//USING the fact that [r/s]^+ equals [-r/s]^+
mid := ((q-1) div 2) + 1;
for a in [mid..q-1] do
table := table cat [table[q-a+1]];
end for;
table := table cat [scale];

return(table);
end function;
```

```
//Multiplies a and b interpreted as a[1] + sqrt(-p)*a[2].
function multiply(a,b,p)
return([a[1]*b[1] - p*a[2]*b[2],a[1]*b[2]+a[2]*b[1]]);
end function;

//Here answer[a] = [alpha^-a]
//n+1 is the maxl power and m is the accuracy
function powers_of_alpha(p,a_p,n,m)
local temp,answer,t,q;

if a_p mod p ne 0 then
temp := a_p mod p;

for k in [2..m] do
t := -(temp^2-a_p*temp+p) div p^(k-1) mod p *
InverseMod(2*temp - a_p,p);
temp := temp + t*p^(k-1);
end for;

q := p^m;
temp := InverseMod(temp,q);
answer := [[temp]];

for k in [2..n+1] do
Append(~answer,[answer[k-1][1]*answer[1][1] mod q]);
end for;

return(answer);
else
if a_p eq 0 then
temp := [0,-1/p];
else
if a_p eq -3 then
temp := [-1/2,-1/6];
else
temp := [1/2,-1/6];
end if;
end if;
answer := [temp];
for k in [2..n+1] do
Append(~answer,multiply(answer[k-1],answer[1],p));
end for;
return(answer);
end if;
end function;

//Here answer = {a} mod p^m
function one_teich(a,p,m)
local answer,t;
```

```
answer := a mod p;
for k in [2..m] do
t := 0;
while (answer + t*p^(k-1))^(p-1) mod p^k ne 1 do
t := t+1;
end while;
answer := answer + t*p^(k-1);
end for;

return(answer);
end function;

//Here answer[a] = {a} (mod p^m)
function list_of_teich(p,m)
local answer;

answer := [];
for a in [1..p-1] do
Append(~answer,one_teich(a,p,m));
end for;
return(answer);
end function;

//Here answer[a] = gamma^(a-1)
//SO THERE IS A SHIFT
//n is the maxl power and m is the accuracy
function powers_of_gamma(gamma,p,n)
local answer;

answer := [1];
for k in [2..p^(n-1)] do
Append(~answer,answer[k-1]*gamma mod (p^n));
end for;

return(answer);
end function;


function lift_polynomial(f)
return(PolynomialRing(Rationals()) ! (PolynomialRing(Integers()) ! f));
end function;

//Here answer[a]=(1+T)^(a-1)
//SO THERE IS A SHIFT
//n is the maxl power and m is the accuracy
function powers_of_oneplusT(p,n,m)
local R,S,answer,q;

q := p^m;
R := ResidueClassRing(q);
```

```
S<T> := PolynomialRing(R);
answer := [1];
for k in [2..p^(n-1)] do
answer := answer cat [answer[k-1]*(1+T)] ;
end for;

return(answer);
end function;

function alpha(n,ALPHA)
   return(ALPHA[n]);
end function;

function modsym(a,MODSYM)
return(MODSYM[a+1]);
end function;

function oneplusT(j,POWER)
return(POWER[j+1]);
end function;

function teich(a,TEICH)
return(TEICH[a]);
end function;

function gamma(j,GAMMA)
return(GAMMA[j+1]);
end function;

function scale(list,c)
local k,answer;

answer:=[];
for k in [1..#list] do
answer := answer cat [list[k]*c];
end for;
return(answer);
end function;

function add(list1,list2)
local k,answer;

if #list1 ne #list2 then
print "Error in add";
end if;

answer:=[];
for k in [1..#list1] do
answer := answer cat [list1[k]+list2[k]];
  end for;
```

```
return(answer);
end function;

function subtract(list1,list2)
local k,answer;

if #list1 ne #list2 then
print "Error in subtract";
end if;

answer:=[];
for k in [1..#list1] do
answer := answer cat [list1[k]-list2[k]];
  end for;

return(answer);
end function;

function measure(a,p,n,ALPHA,MODSYM)
local q,q1;

q := p^n;
q1 := q div p;
return( subtract(scale(alpha(n,ALPHA),modsym(a,MODSYM)),
   scale(alpha(n+1,ALPHA),modsym((a mod q1) * p ,MODSYM))));

end function;

//This procedure returns a polynomial approximation to the p-adic
//L-series of the elliptic curve corresponding to V twisted by a quadratic
//character of discriminant D.  The degree of the polynomial is p^n-1.
//If a_p is prime to p (ordinary) then the answer is simply a polynomial.
//Otherwise (in the supersingular case) the answer is in the form [G,H]
//where the L-series is approximated by G + sqrt(-p) * H.
//
//Input: V - space of modular symbols
// p - prime
// a_p - eigenvalue of T_p acting on V
// n - relates to degree of approximation
// D - value to twist by
function Lseries(V,p,a_p,n,D)
local answer,q;

ALPHA := powers_of_alpha(p,a_p,n,n+1);
MODSYM := table_of_modular_symbols(V,p,n,D);
POWER := powers_of_oneplusT(p,n,n+1);
TEICH := list_of_teich(p,n+1);
GAMMA := powers_of_gamma(1+p,p,n+1);
```

```
q1 := p^(n-1);
q := q1*p;
if a_p mod p ne 0 then
answer := [0];
else
answer := [0,0];
end if;

for a in [1..p-1] do
for j in [0..q1-1] do
answer := add(answer,
                 scale(measure(teich(a,TEICH)*gamma(j,GAMMA) mod
                               q,p,n,ALPHA,MODSYM),
                        lift_polynomial(oneplusT(j,POWER))));
end for;
end for;

answer:=scale(answer,1/D);
answer:=scale(answer,1/MODSYM[#MODSYM]);

return(answer);
end function;

function highest_power(n,p)
local k,temp;

k:=0;
temp:=1;
while (n ge temp) do
temp := temp*p;
k := k+1;
end while;
return(k-1);
end function;

function valuation_of_coefficients(f,p,n)
local answer,j,temp,okay,bound,list_bad;

answer := [];
okay:=true;
list_bad:=[];
for j in [1..Degree(f)+1] do
if Coefficient(f,j-1) ne 0 then
answer := answer cat [Valuation(Coefficient(f,j-1),p)];
else
answer := answer cat [10000];
end if;

bound := n-1-highest_power(j-1,p);
if (Valuation(Coefficient(f,j-1),p) ge bound) and (j ne 1) then
```

```
list_bad := list_bad cat [j];
end if;
end for;

if f eq 0 then
answer := [];
answer := answer cat [10000] cat [10000] cat [10000];
end if;

return(<answer,list_bad>);
end function;

function newton_polygon(C)
local k,here,next,min_slope,answer,okay,prev_slope;

answer:= [];
okay:=true;
here:=1;
next:=1;

while here lt #C do
min_slope:=100000;
for k in [here+1..#C] do
if (C[k]-C[here])*(1.0)/(k-here) le 1.0*min_slope then
min_slope := (C[k]-C[here])/(k-here);
next := k;
end if;
end for;
answer := answer cat [[next-here,-min_slope]];
here := next;
end while;
return(answer);
end function;

function find_last_nontriv(NP,p,sign)
local slope,place;

place := 0;
if sign mod 2 eq 0 then
slope := 1/(p*(p-1));
else
slope := 1/(p-1);
end if;

for k in [1..#NP] do
if (NP[k][2] gt 0) then
if (NP[k][2] ne slope) or (NP[k][1] ne 1/slope) then
place:=k;
else
slope:=slope/(p^2);
```

```
end if;
end if;
end for;

return(place);
end function;


function refine_newton_polygon(NP,p,a_p,sign)
local answer,k,last;

answer := [];
if a_p mod p ne 0 then
for k in [1..#NP] do
if NP[k][2] gt 0 then
answer := answer cat [NP[k]];
end if;
end for;
else
last := find_last_nontriv(NP,p,sign);
for k in [1..last] do
answer := answer cat [NP[k]];
end for;
end if;

return(answer);
end function;

function minimize_ord(C,p,n)
local k,start;

start:=1;
if C[1] eq 10000 then
start := 2;
end if;

for k in [start..#C] do
if C[k] ge n-1-highest_power(k-1,p) then
C[k] := n-1-highest_power(k-1,p);
end if;
end for;
return(C);
end function;

function in_list(list,num)
local k,flag;

flag:=false;
for k in [1..#list] do
if list[k] eq num then
```

```
flag := true;
end if;
end for;

return(flag);
end function;

function compute_rank(C,N,r,D)
local rank;

if C[1] eq 10000 then
rank := 1;
else
rank :=0;
  end if;

if (r eq 0) and (rank eq 1) and (KroneckerSymbol(D,N) eq 1) then  //it has even rank >= 2.
rank:=2;
C[2]:=10000;
end if;

if (r eq 1) and (rank eq 1) and (KroneckerSymbol(D,N) eq -1) then  //it has even rank >= 2.
rank:=2;
C[2]:=10000;
end if;

return(<C,rank>);
end function;


function check_NP(NP,C,list_bad,rank,p,a_p,n,sign)
local temp,val,k,good;

//checks if break points are accurate.

good:=true;
temp := refine_newton_polygon(NP,p,a_p,sign);
val := 1;
for k in [1..#temp] do
val := val + temp[k][1];
if (in_list(list_bad,val)) then
good := false;
print "Failed at a breakpoint,",val;
end if;
end for;

//checks if lowering the valuation of any unknown coefficient to its
//minimal possible value affects the newton polygon.

C_min := minimize_ord(C,p,n); //lowers all unknown coefficients to lowest possible valuation
```

```
if rank eq 2 then
    C_min[2] := 10000;
end if;

NP_min := newton_polygon(C_min);
if refine_newton_polygon(NP_min,p,a_p,sign) ne refine_newton_polygon(NP,p,a_p,sign) then
good := false;
print "Failed while lowering.";
end if;

return(good);
end function;

function lambda(NP)
local k,lam;

k:=1;
lam := 0;
while (k le #NP) and (NP[k][2] gt 0) do
   lam := lam + NP[k][1];
k := k+1;
end while;

return(lam);
end function;

function mu(NP,C,rank)
local k,val,m,start;

k:=1;
val := 0;
if (rank ne 2) then
while (k le #NP) and (NP[k][2] gt 0) do
if NP[k][2] lt 1000 then
val := val + NP[k][1]*NP[k][2];
end if;
k := k+1;
end while;
else
if (#NP gt 0) and (NP[1][1] eq 1) then
start := 3;
else
if #NP eq 0 then
start := 1;
else
start := 2;
end if;
end if;
k:=start;
```

```
while (k le #NP) and (NP[k][2] gt 0) do
if NP[k][2] lt 1000 then
val := val + NP[k][1]*NP[k][2];
end if;
k := k+1;
end while;
end if;

if rank eq 0 then
m := C[1]-val;
else
if rank eq 1 then
m := C[2] - val;
else
m := C[3] - val;
end if;
end if;

return(m);

end function;

function remove_two(list)
local answer,k;

answer := [];

for k in [1..#list] do
if list[k] ne 2 then
answer := answer cat [list[k]];
end if;
end for;

return(answer);

end function;

function leading_term(C,rank)

if rank eq 0 then
return(C[1]);
else
if rank eq 1 then
return(C[2]);
else
if rank eq 2 then
return(C[3]);
end if;
end if;
end if;
```

```
end function;

function compute_untwisted_rank(V,p,a_p)
local L,C;

L:=Lseries(V,p,a_p,2,1);
L:=L[1];
C := valuation_of_coefficients(L,p,1);
if C[1][1] ne 10000 then
return(0);
else
return(1);
end if;
end function;

function remove_unit_zeroes(NP)
local k,answer;

answer := [];
for k in [1..#NP] do
if NP[k][2] gt 0 then
answer := answer cat [NP[k]];
end if;
end for;
return(answer);
end function;

//Has the space of modular symbols already pre-computed
function iwasawa_invariants_w_curve_ord(N,V,p,a_p,n,D)
local C,NP,k,lam,m,rank,j,list_bad,list_breapoints,C_min,NP_min,val,good,m2,lam2,temp,leading

L:=Lseries(V,p,a_p,n,D)[1];
C := valuation_of_coefficients(L,p,n);
list_bad:=C[2];
C:=C[1];

//FIRST COMPUTE RANK

rank := compute_untwisted_rank(V,p,a_p);
C := compute_rank(C,N,rank,D);

rank := C[2];  //changes valuation of a_1 to 10000 if guesses rank 2.
C := C[1];

if rank eq 2 then
list_bad := remove_two(list_bad);
end if;

NP := newton_polygon(C);
good := check_NP(NP,C,list_bad,rank,p,a_p,n,0);  //0 represents nothing.
```

```
NP := remove_unit_zeroes(NP);
leading := leading_term(C,rank);

if (leading eq 0) and (in_list(list_bad,rank+1) eq false) then
   good:=true;
end if;

lam:=lambda(NP);
m := mu(NP,C,rank);

return(<rank,leading,m,lam,NP,good>);
end function;

function scale_by_p(C)
local k;

for k in [1..#C] do
if C[k] ne 10000 then
C[k]:=C[k]+1;
end if;
end for;

return(C);
end function;


function remove_forced_zeroes(NP,p,init_slope)
local k,slope,temp;

temp := [];
k := 1;
slope:=init_slope;
while (k le #NP) and (NP[k][2] gt 0)  do
if NP[k][2] ne slope then
temp := temp cat [NP[k]];
else
if NP[k][1] - 1/slope ne 0 then
temp := temp cat [[NP[k][1] - 1/slope,slope]];
end if;
  slope := slope/(p^2);
end if;
k := k+1;
end while;

return(temp);
end function;

//Has the space of modular symbols already pre-computed
function iwasawa_invariants_w_curve_ss(N,V,p,a_p,n,D)
local L,G,H,k,CG,CH,mg,mh,lamg,lamh,NPG,NPH,rank,rankg,rankh,
```

```
list_badg,list_badh,denomg,denomh,goodg,goodh,list_breakpointsg,list_breakpointsh,
leadingg,leadingh;

L := Lseries(V,p,a_p,n,D);
G := L[1];
H := L[2];

// denomg and denomh represent the number of p's in the denominator of the measure

if n mod 2 eq 0 then
denomg:=n div 2;
denomh:=(n+2) div 2;
else
denomg:=(n+1) div 2;
denomh:=(n+1) div 2;
end if;

CG := valuation_of_coefficients(G,p,n-denomg);
CH := valuation_of_coefficients(H,p,n-denomh);

list_badg := CG[2];
list_badh := CH[2];

CG:=CG[1];
CH:=CH[1];

//Computing the rank

rank:=compute_untwisted_rank(V,p,a_p);

CG := compute_rank(CG,N,rank,D); //changes a_1 if guesses rank 2
rankg := CG[2];
CG := CG[1];

CH := compute_rank(CH,N,rank,D);
rankh := CH[2];
CH := CH[1];

if rankg ne rankh then
print "WEIRD..ranks different in G and H";
PrintFile("notes","ranks different in G and H");
PrintFile("notes",D);
end if;

rank := rankg;
if rank eq 2 then
list_badg := remove_two(list_badg);
end if;

if rank eq 2 then
```

```
list_badh := remove_two(list_badh);
end if;

NPG := newton_polygon(CG);
NPH := newton_polygon(CH);

goodg := check_NP(NPG,CG,list_badg,rank,p,a_p,n,0); //0 represents g
goodh := check_NP(NPH,CH,list_badh,rank,p,a_p,n,1); //1 represents h

leadingg := leading_term(CG,rank);
leadingh := leading_term(CH,rank);

if (a_p eq 0) then
if (leadingg eq -1) and (in_list(list_badg,rank+1) eq false) then
  goodg:=true;
end if;

if (leadingh eq -1) and (in_list(list_badh,rank+1) eq false) then
  goodh:=true;
end if;
end if;


if a_p eq 0 then
NPG := remove_forced_zeroes(NPG,p,1/((p-1)*p));
NPH := remove_forced_zeroes(NPH,p,1/(p-1));
else
if a_p eq -3 then
NPG := remove_forced_zeroes(NPG,p,1/((p-1)*p));
NPH := remove_forced_zeroes(NPH,p,1/(p-1)*p^2);
end if;
end if;

lamg := lambda(NPG);
lamh := lambda(NPH);

mg := mu(NPG,CG,rank)+1;
mh := mu(NPH,CH,rank)+1;

return(<rank,leadingg,mg,lamg,NPG,goodg,leadingh,mh,lamh,NPH,goodh>);
end function;

// This procedure returns many invariants of the curve corresponding
// to V twisted by D.
//
// In the ordinary case the data returned is:
// <rank,valuation of leading coefficient,mu invariant,lambda invariant,
//  newton polygon of L-series, boolean saying whether the above data
//  are accurate>
//
```

```
// The newton polygon is returned as [n_1,s_1],[n_2,s_2],... where there are
// n_1 roots of slope s_1, etc.
//
// In the supersingular case the data returned is similar.  The first entry is
// the rank, then the next five are the data for my "g" iwasawa function and
// then the next five for my "h" iwasawa function.

function iwasawa_invariants_w_curve(N,V,p,a_p,n,D)
if a_p mod p ne 0 then
return(iwasawa_invariants_w_curve_ord(N,V,p,a_p,n,D));
else
return(iwasawa_invariants_w_curve_ss(N,V,p,a_p,n,D));
end if;
end function;
```