Appendix §A. Sage code

The code we used to compute the data in Section 6 is given below. The code from line 1 to line 43 is taken from [Pol], which sets up the relevant polynomial rings and compute a list of Mazur–Tate elements. The code from line 44 to line 115 is new, which computes approximations to $L_p^{\pm}$, the corresponding λ-invariants and the slopes of the roots. The code from line 116 to line 167 is taken from [Pol], except for lines 142 to 146, where the twist formula given in Section 2 is utilized to compute modular symbols for quadratic twists. The lines 168 to 200 are again from [Pol]. The rest of the code is new.

```
1  import time
2  from sage.databases.cremona import parse_cremona_label
3  import numpy as numpy
4  
5  #this is the final code used for the computation
6  
7  def invariants_of_ec(E,p,D = 1):
8    """
9    Returns the mu and lambda invariants, and the valuations of the roots of the +/-
       p-adic L-function of E.
10   Input:
11   -‘‘E‘‘ -- elliptic curve
12   -‘‘p‘‘ -- prime, this code works only for odd primes
13     - ‘‘D‘‘ -- The discriminant of the quadratic field being twisted by, 1 by
     default, eg if the field is Q(\sqrt(-23)), then D = -23, if
14   it is Q(\sqrt(-21)), then enter D = -84
15     Output:
16   In the supersingular case: (mu_plus,mu_minus), [lambda_plus, valuations of roots
       of Lp+,lambda_minus, valuations of roots of Lp-]
17   If the constant terms of L_p^+, L_p^- are non-vanishing then the valuations of
     the roots obtained are exact, as proven in the article
18   However, if the constant term vanishes, then the valuations of the roots
     returned might not be exact, and further approximations
19      may need to be computed to obtain the exact valuations.
20   In the examples presented in the article, the constant term is always non-zero
     and hence the valuations obtained are exact.
21   Note that the forced zero at 0 due to the functional equation does not show up
     in list of valuations of roots
22   If the minimal mu wasn’t found to be, then mu’s are returned as ’?’
23   """
24     #This code builds upon Pollack’s code for the Iwasawa invariants in the
     supersingular case.
25   Etwist = E.quadratic_twist(D)
26   if Etwist.is_supersingular(p): #checks if E is supersingular at p
27     r = Etwist.analytic_rank()
28     correction = 0
29     if (E.quadratic_twist(D)).root_number() == -1:
30       correction = 1 # This accounts for the forced zero at T = 0 due to the
     functional equation
31     MTs = [MazurTate(E,p,1,D), MazurTate(E,p,2,D)] # Keeps track of Mazur-Tate
     elements
32     done = (mu(MTs[0],p) == 0) and (mu(MTs[1],p) == 0)
33     n = 3
```

```python
  while not done and (n <= mu_bail(p)): #Computes the level at which the mu invariant vanishes for L_{p,n}^{+-}
    MTs += [MazurTate(E,p,n,D)]
    done = (mu(MTs[n-1],p) == 0) and (mu(MTs[n-2],p) == 0)
    n = n + 1
  if done == 0:
    print("mu-invariant does not vanish")
  n = n - 1
  Qp = pAdicField(p,2*n+5)
  S.<T> = PolynomialRing(Qp)
  R.<T> = PolynomialRing(QQ)
  def approx(poly,m):
        #This function computes approximations to Lp +/- and computes their newton slopes
    if m%2 == 1:
      quotient = (poly.factor()/Phip(m,p)).expand()
      return R(quotient)
    if m%2 == 0:
      quotient = (poly.factor()/Phim(m,p)).expand()
      return R(quotient)
  def lambdamw(quotient): #This computes the lambda_III
    R.<T> = PolynomialRing(QQ) ; quotient = R(quotient)
    l = []
    for j in range(1,lamb(quotient,p)+1):
      if cyc(j,p).divides(quotient):
        l += (S(cyc(j,p))).newton_slopes()
    return l
  #The next piece of code computes the approximations to the +/- p-adic L-functions, I use the last two mazur tate elements computed. This part of the code is new
  if n%2 == 0:
    lambda_plus = lamb(MTs[n-2],p)-qn(p,n-1)
    lambda_minus = lamb(MTs[n-1],p)-qn(p,n)
    if correction == 1 and lambda_plus == 1 and lambda_minus == 1:
      return (0,0),[0,1,[],0,1,[]]
    Lpplus = (-1)^(n/2)*approx(MTs[n-2],n-1)
    Lpminus = (-1)^(n/2)*approx(MTs[n-1],n)
    k = needed(p,lambda_plus, lambda_minus)
    if Lpplus[0]!= 0 and Lpminus[0] != 0: #checks if constant term is non-zero, if it is, then we compute approximations to the required level
      Nplus = k[0] + 2*Lpplus[0].valuation(p)
      Nminus = k[1] + 2*Lpminus[0].valuation(p)
      enough = n >= Nplus and n >= Nminus
      while not enough:
        n = n + 1
        if n%2 == 0 and n < Nminus + 1:
          Lpminus = approx(MazurTate(E,p,n,D),n)
        elif n%2 == 1 and n < Nplus + 1:
          Lpplus = approx(MazurTate(E,p,n,D),n)
        Nplus = k[0] + 2*Lpplus[0].valuation(p)
        Nminus = k[1] + 2*Lpminus[0].valuation(p)
        enough = n >= Nplus and n >= Nminus
    slopes_plus = [i for i in S(Lpplus).newton_slopes() if i > 0]
```

```
      lambdamw_plus = len(lambdamw(Lpplus)) + correction
      lambdaIII_plus = lambda_plus - lambdamw_plus
      slopes_minus = [i for i in S(Lpminus).newton_slopes() if i > 0]
      lambdamw_minus = len(lambdamw(Lpminus)) + correction
      lambdaIII_minus = lambda_minus - lambdamw_minus
    else:
      lambda_minus = lamb(MTs[n-2],p)-qn(p,n-1)
      lambda_plus = lamb(MTs[n-1],p)-qn(p,n)
      if correction == 1 and lambda_plus == 1 and lambda_minus == 1:
        return (0,0),[0,1,[],0,1,[]]
      Lpplus = (-1)^((n+1)/2)*approx(MTs[n-1],n)
      Lpminus = (-1)^((n+1)/2)*approx(MTs[n-2],n-1)
      k = needed(p,lambda_plus, lambda_minus)
      if Lpplus[0] != 0 and Lpminus[0] != 0: #checks if constant term is non-zero, if it is, then we compute approximations to the required level
        Nplus = k[0] + 2*Lpplus[0].valuation(p)
        Nminus = k[1] + 2*Lpminus[0].valuation(p)
        enough = n >= Nplus and n >= Nminus
        while not enough and Lpplus[0] !=0 and Lpminus[0] != 0:
          n = n + 1
          if n%2 == 0 and n < Nminus + 1:
            Lpminus = approx(MazurTate(E,p,n,D),n)
          elif n%2 == 1 and n < Nplus + 1:
            Lpplus = approx(MazurTate(E,p,n,D),n)
          Nplus = k[0] + 2*Lpplus[0].valuation(p)
          Nminus = k[1] + 2*Lpminus[0].valuation(p)
          enough = n >= Nplus and n >= Nminus
      slopes_plus = [i for i in S(Lpplus).newton_slopes() if i > 0]
      lambdamw_plus = len(lambdamw(Lpplus)) + correction
      lambdaIII_plus = lambda_plus - lambdamw_plus
      slopes_minus = [i for i in S(Lpminus).newton_slopes() if i > 0]
      lambdamw_minus = len(lambdamw(Lpminus)) + correction
      lambdaIII_minus = lambda_minus - lambdamw_minus
    return (0,0), [lambda_plus, slopes_plus,lambda_minus, slopes_minus]
  else : raise ValueError(’the elliptic curve(or its twist if D != 1) is not supersingular at the prime p’)

def MazurTate(E,p,n,D = 1):
  """
  This code is mostly as in Rob Pollack’s github page.
 The only modification is a speedup while computing the modular symbols for the twists.
 Returns the p-adic Mazur-Tate element of level n.  That is, for p odd, we take the element
    sum_{a in (Z/p^{n+1}Z)^*} [a/p^{n+1}]^+_E sigma_a
  in Q[Gal(Q(mu_{p^{n+1}}))] and project it to Q[Gal(Q_n/Q)] where Q_n is the
  n-th level of the cyclotomic Z_p-extension.  The projection here is twisted by omega^twist so
  that the group algebra element [a] maps to omega^twist(a)[a].
  (For p=2, one projects from level n+2, see Kurihara-Otsuki)
  Input:
  - ‘‘E‘‘ -- elliptic curve
  - ‘‘p‘‘ -- prime
  - ``n`` -- integer >= -1
  - D the discriminant of the quadratic field we are twisting by
 """
  start = time.time()
  if D > 0:
    M1 = E.modular_symbol()
  else:
    M1 = E.modular_symbol(sign = -1)
  def twisted_modularsymbol(r):
    answer = 0
    for a in Zmod(abs(D)).list_of_elements_of_multiplicative_group():
      answer = answer + kronecker(D,a)*M1(r+a/abs(D))
    return answer
  R.<T> = PolynomialRing(QQ)
  if n > 0:
    mt = R(0)
    if p > 2:
      gam = 1+p
      ## should check carefully the accuracy needed here
      Qp = pAdicField(p,2*n+5)
      teich = Qp.teichmuller_system()
      teich = [0] + teich  ## makes teich[a] = omega(a)
      teich = [ZZ(teich[a]) for a in range(p)]
      gampow = 1 ## will make gampow = gam^pow
      oneplusTpow = 1 ## will make oneplusTpow = (1+T)^pow
      for j in range(p^(n)):
        cj = sum([twisted_modularsymbol(gampow * teich[a] / p^(n+1)) for a in range(1,(p+1)/2)])
        mt = mt + R(cj) * oneplusTpow
        gampow = gampow * gam
        oneplusTpow = oneplusTpow * (1 + T)
      end = time.time()
      t = end-start
      ans = 2*mt
  return ans

def mu(f,p):
  """Returns the (p-adic) mu-invariant of f"""
  if f == 0:
    return oo
  else:
    return min([f[a].valuation(p) for a in range(f.degree()+1)])

def mu_bail(p):
  """
  For a given prime p, returns the n for which we should keep trying to
  compute Mazur-Tate elements to level p^n in hoping that their mu-invariants will vanish.
  The below values were just picked after a few trial runs.  In practice,
  mu should always eventually be zero (by how the code is normalized) and
  so these parameters are just setting how long we want to wait
  Input:
  - ``p`` -- prime
  """
 # This is same as in [Pol]
  if p<1000:
    return max(round(log(1000)/log(p)),4)-1
  else:
    return -1

def lamb(f,p):
  """Returns the (p-adic) lambda-invariant of f"""
    # This is same as in [Pol]
  if f == 0:
    return oo
  else:
    m = mu(f,p)
    v = [f[a].valuation(p) for a in range(f.degree()+1)]
    return v.index(m)

def qn(p,n):
  """"q_n as defined by Kurihara"""
    # This is same as in [Pol]
  if n%2 == 0:
    return sum([p^a - p^(a-1) for a in range(1,n,2)])
  else:
    return sum([p^a - p^(a-1) for a in range(2,n,2)])

def cyc(n, p):
    #This is new (not from Pollack's code)
    #creates the cyclotomic polynomial Phi_n(poly)
    R.<T> = PolynomialRing(QQ)
    ans = R(0)
    for j in range(p):
        ans = ans + (1+T)^(p^(n-1)*j)
    return ans

def Phip(n,p):
    #This is new (not from Pollack's code)
    #computes the product of phi_j(poly) for even j less than or equal to n
    R.<T> = PolynomialRing(QQ)
    ans = R(1)
    for j in [2*i for i in range(1,integer_floor((n+2)/2))]:
        ans = ans * cyc(j, p)
    return ans

def Phim(index,p):
    #This is new (not from Pollack's code)
    #computes the product of phi_j(poly), for odd j less than or equal to n
    R.<T> = PolynomialRing(QQ)
    ans = R(1)
    for j in [2*i - 1 for i in range(1,integer_floor((index+3)/2))]:
        ans = ans * cyc(j, p)
    return ans

def needed(p,d1,d2):
    """This is new, and it computes the N needed, as in Lemma 1"""
    done = 0
    n = 1
    check1 = (p^(n+1) - 1)/(p+1) > d1
    while check1 == 0:
        n = n + 2
        check1 = (p^(n+1) - 1)/(p+1) > d1
    l = [n]
    n = 2
    check2 = p*(p^n - 1)/(p+1) > d2
    while check2 == 0:
        n = n + 2
        check2 = p*(p^n - 1)/(p+1) > d2
    l.append(n)
    return l
```

References

[Pol] Robert Pollack. https://github.com/rpollack9974/Iwasawa-invariants/blob/master/IwInv.sage.

Sai Sanjeev Balakrishnan
Department of Mathematics, University of California Berkeley, Berkeley, California, United States
Email address: `saisanjeev@berkeley.edu`

Antonio Lei
Department of Mathematics and Statistics, University of Ottawa, 150 Louis-Pasteur Pvt, Ottawa, ON, Canada K1N 6N5
Email address: `antonio.lei@uottawa.ca`

Bharathwaj Palvannan
Department of Mathematics, Indian Institute of Science, Bangalore - 560012, India
Email address: `bharathwaj@iisc.ac.in`