Metallic behaviour in SOI quantum well with strong intervalley scattering - Supplementary information

V. T. Renard[1*], I. Duchemin[2], Y. Niida[3], A. Fujiwara[4], Y. Hirayama[3], and K. Takashina[5]
[1]*SPSMS, UMR-E 9001, CEA-INAC/UJF-Grenoble 1, INAC, Grenoble F-38054, France*
[2]*SP2M, UMR-E 9001, CEA-INAC/UJF-Grenoble 1, INAC, Grenoble F-38054, France*
[3]*Graduate School of Science, Tohoku University, 6-3 Aramakiaza Aoba, Aobaku, Sendai, 980-8578 Japan*
[4]*NTT Basic Research Laboratories, NTT Corporation, Atsugi-shi, Kanagawa 243-0198, Japan*
[5]*Department of Physics, University of Bath, Bath BA2 7AY, UK*

This document contains the supplemental material on the analysis of Weak Localization magneto-conductance.

I. WEAK LOCALIZATION IN PRESENCE OF INTERVALLEY SCATTERING.

The theory of weak localization in two-valley systems in presence of intervalley scattering and for small valley splitting ($\Delta_v < \hbar/\tau$) is developed in Ref. 1. The weak localization to conductivity has the following form:

$$\delta\sigma_{WL}(B,T) = \Delta\sigma^{(a)} + \Delta\sigma^{(b)} \tag{1}$$

with

$$\Delta\sigma^{(a)} = -\frac{e^2\,b}{2\pi^2\hbar}\sum_{N=0}^{\infty}\mathcal{C}_N P_N^2\,, \tag{2}$$

$$\Delta\sigma^{(b)} = \frac{e^2\,b}{2\pi^2\hbar}\sum_{N=0}^{\infty}\left(\mathcal{C}_N + \mathcal{C}_{N+1}\right)Q_N^2/2\,. \tag{3}$$

$$\mathcal{C}_N = \frac{2(1-\tau/\tau_\perp)^3\,P_N}{1-(1-\tau/\tau_\perp)P_N} + \frac{P_N}{1-P_N} - \frac{(1-2\tau/\tau_\perp)^3 P_N}{1-(1-2\tau/\tau_\perp)P_N}\,, \tag{4}$$

where P_N and Q_N are coefficients which are given by

$$(1+z)P_N = \mathcal{P}_N = s\int_0^{\infty} dx\,\mathrm{L}_N(x^2)e^{\left(-sx-\frac{x^2}{2}\right)}\,, \tag{5}$$

$$(1+z)Q_N = \mathcal{Q}_N = s\int_0^{\infty} dx\,x\frac{\mathrm{L}_N^1(x^2)}{\sqrt{N+1}}e^{\left(-sx-\frac{x^2}{2}\right)}\,, \tag{6}$$

and L_N and L_N^1 are the Laguerre polynomials, $\tau_\perp$ the intervalley scattering time, z is the ratio of the transport time τ and the phase coherence time τ_φ and $s = (1+z)\sqrt{\frac{2}{b}}$ with $b = 2el^2B/\hbar$ (l is the mean free path).

Kawabata[2] showed that the P_N can be found recursively as follows:

$$\mathcal{P}_0 = s\sqrt{\frac{\pi}{2}}e^{\frac{s^2}{2}}\,\mathrm{erfc}\left(\frac{s}{\sqrt{2}}\right) \tag{7}$$

$$\mathcal{P}_1 = s^2 - s^2\mathcal{P}_0 \tag{8}$$

$$\mathcal{P}_N = \frac{1}{N}\left[(N-1+s^2)(\mathcal{P}_{N-2}-\mathcal{P}_{N-1}) + (N-2)\mathcal{P}_{N-3}\right] \tag{9}$$

* Correspondance to : vincent.renard@cea.fr

Once the $\mathcal{P}_N$ are computed, the $\mathcal{Q}_N$ terms can be evaluated with the following expression[3]:

$$\mathcal{Q}_N = \frac{s}{\sqrt{N+1}}(1-\mathcal{P}_N) - \sqrt{\frac{N}{N+1}}\mathcal{Q}_{N-1} \tag{10}$$

Contrary to Refs. 1, 3 we used the recursive definition of P_N and Q_N for all N up to $N = 10^5$. The stability of the recursion on P_N is guaranted by working with large precision numbers. In particular, lower b values require increased numbers of digits. Therefore our implementation makes use the multiple precision real number class (through the C++ interface for MPFR library provided by Pavel Holoborodko. The following part presents the code used for the computation.

A. Main file

```
#include <math.h>
#include <stdlib.h>
#include <iostream>
#include "mpreal.h"
#include "Compute_Pn.h"
#include "Compute_Qn.h"
#include "Compute_Cn.h"
#include "Compute_Sigma.h"

using namespace mpfr;

int usage(int argc, char **argv)
{
  // print explanations
  std::cout << "usage: " << argv[0] << " nmax prec b tau/tau_v tau/tau_phi" << std::endl;
  std::cout << std::endl;
  std::cout << " nmax        : maximum order of Pn/Qn devellopment" << std::endl;
  std::cout << " prec        : precision (in bits) to be used for Pn's recursion calculation" << std::endl;
  std::cout << " b           : reduced magnetic field" << std::endl;
  std::cout << " tau/tau_v   : transport to intervalley scattering time ratio" << std::endl;
  std::cout << " tau/tau_phi : transport to phase coherence time ratio" << std::endl;
  std::cout << std::endl;
  std::cout << "returns: " << std::endl;
  std::cout << std::endl;
  std::cout << " sigma_a,sigma_b : weak localization detailed contributions (see A. Y. Kuntsevich et al.)" << std::endl;
  std::cout << " d_sigma         : sigma_a + sigma_b, in e/h unit" << std::endl;
  std::cout << std::endl;
  // return
  return 1;
}

int main(int argc, char **argv)
{
  // print info and refs
  std::cout << std::endl;
  std::cout << "================================================================================" << std::endl;
  std::cout << "= compute weak localization correction with intervalley scattering             =" << std::endl;
  std::cout << "= see A. Y. Kuntsevich et al., Phys Rev. B. 75, 195330 (2007) for further references =" << std::endl;
  std::cout << "=                                                                              =" << std::endl;
  std::cout << "= Authors: I.Duchemin, V.T.Renard                                 04/11/2012 =" << std::endl;
  std::cout << "================================================================================" << std::endl;
  std::cout << std::endl;

  // check arguments
  if ( argc!=6 ) return usage(argc,argv);

  // get inputs
  int nmax=atoi(argv[4]);
  int prec=atoi(argv[5]);

  // inform on parameters
  std::cout << "using nmax =" << nmax << std::endl;
  std::cout << "using prec =" << prec << std::endl;

  // set output format
  std::cout.precision(5);
  std::cout.setf(std::ios::scientific);

  // set extended precision
  mpreal::set_default_prec(prec);

  // get remaining inputs
```

```
63    double b=atof(argv[1]);
64    double t_over_tv=atof(argv[2]);
65    double t_over_tphi=atof(argv[3]);
66
67    // inform on parameters
68    std::cout << "using b      =" << b << std::endl;
69    std::cout << "using tau/tau_v   =" << t_over_tv << std::endl;
70    std::cout << "using tau/tau_phi =" << t_over_tphi << std::endl;
71
72    // compute s in multiple precision
73    mpreal s=sqrtl(2.0/b)*(1.0+t_over_tphi);
74
75    // compute Pn in extended precision
76    std::vector<mpreal> Pn_ext=compute_Pn(s,nmax);
77
78    // set Pn in double precision
79    std::vector<double> Pn(nmax);
80    for ( int i=0 ; i<nmax ; i++ )
81    {
82      Pn[i]=Pn_ext[i].toDouble();
83    }
84
85    // visual check for Pn's values
86    std::cout << std::endl;
87    std::cout << "Pn[0]=" << Pn[0] << std::endl;
88    std::cout << "Pn[1]=" << Pn[1] << std::endl;
89    std::cout << "Pn[$]=" << Pn.back() << std::endl;
90    std::cout << std::endl;
91
92    // compute Qn in double precision
93    std::vector<double> Qn=compute_Qn(s.toDouble(),Pn);
94
95    // visual check for Qn's values
96    std::cout << "Qn[0]=" << Qn[0] << std::endl;
97    std::cout << "Qn[1]=" << Qn[1] << std::endl;
98    std::cout << "Qn[$]=" << Qn.back() << std::endl;
99    std::cout << std::endl;
100
101   // compute Cn in double precision
102   std::vector<double> Cn=compute_Cn(Pn,t_over_tv,t_over_tphi);
103
104   // compute sigma_a and sigma_b in double precision
105   double sigma_a=compute_sigma_a(Pn,Qn,Cn,b,t_over_tphi);
106   double sigma_b=compute_sigma_b(Pn,Qn,Cn,b,t_over_tphi);
107
108   // display results in units of e^2/h
109   std::cout << "results (e^2/h unit):" << std::endl;
110   std::cout << "sigma_a=" << -sigma_a  << std::endl;
111   std::cout << "sigma_b=" <<  sigma_b  << std::endl;
112   std::cout << std::endl;
113   std::cout << "b= " << b << "   d_sigma= " << sigma_b-sigma_a << std::endl;
114   std::cout << std::endl;
115
116   return 1;
117 }
```

B. Function to compute $P'_N s$

```
1  /*! \file Compute_Pn.h
2      \brief Template function for computing Pn's values with desired precision
3  */
4
5  #ifndef COMPUTEPN_H
6  #define COMPUTEPN_H
7
8  #include <cmath>
9  #include <vector>
10 #include <iostream>
11
12 //! \brief recursive function for Pn(s) computation up to order n. The 1/(1+z) normalization is not applied here.
13 ///
14 /// define Pn(s)= s * int exp(-s*x-x^2/2) * Ln(x^2) * dx
15 ///
16 /// \param s real number, the argument of Pn(s)
17 /// \param nmax the maximum order of the development
18 template <class Precision>
19 std::vector<Precision> compute_Pn(Precision s, int nmax)
20 {
21   //
22   // check input
```

```
23    //
24    if (nmax<0)
25    {
26      std::cout << "Error: nmax argument should be > 0. - nmax=" << nmax << " detected in compute_Pn -" << std::endl;
27    }
28    //
29    // allocate memory
30    //
31    std::vector<Precision> Pn(nmax);
32    //
33    // avoid useless arithmetic...
34    //
35    Precision s2=s*s;
36    //
37    // try to get precise constants
38    //
39    Precision ONE=1.0;
40    Precision TWO=2.0;
41    Precision PI =4.0*atan(ONE);
42    Precision sqrt2=sqrt(TWO);
43    Precision sqrtPIover2=sqrt(PI/TWO);
44    //
45    // get P0 with high precision
46    //
47    Pn[0]=s*sqrtPIover2*exp(s2/2.0)*erfc(s/sqrt2);
48    //
49    // get P1 from P0
50    //
51    Pn[1]=s2*(1.0-Pn[0]);
52    //
53    // get P2
54    //
55    Pn[2]=(1.0+s2)/(2.0)*(Pn[0]-Pn[1]);
56    //
57    // iterate recurence
58    //
59    for ( int i=3 ; i<nmax ; i++ )
60    {
61      //
62      // compute following value
63      //
64      Precision n=i-TWO;
65      Pn[i]=((n+1.0+s2)/(n+2.0))*(Pn[i-2]-Pn[i-1])+(n/(n+2.0))*Pn[i-3];
66    }
67    //
68    // return result
69    //
70    return Pn;
71  }
72
73  #endif
```

C. Function to compute $Q'_N s$

```
1   /*! \file Compute_Qn.h
2       \brief Template function for computing Qn's values with desired precision from Pn's values
3   */
4
5   #ifndef COMPUTEQN_H
6   #define COMPUTEQN_H
7
8   #include <cmath>
9   #include <vector>
10
11  //! \brief recursive function for Qn(s) computation up to order n. The 1/(1+z) normalization is not applied here.
12  ///
13  /// define Qn(s)= s/sqrt(n+1) * int exp(-s*x-x^2/2) * x * L_n^1(x^2) * dx
14  ///
15  /// \param s real number, the argument of Qn(s)
16  /// \param Pn Pn(s) values up to order n
17  template <class Precision>
18  std::vector<Precision> compute_Qn(Precision s, std::vector<Precision> &Pn)
19  {
20    //
21    // get number of elements to be computed
22    //
23    int nmax=Pn.size();
24    //
25    // allocate memory
26    //
```

```
27    std::vector<Precision> Qn(nmax);
28    //
29    // get Q0
30    //
31    Qn[0]=s*(1.0-Pn[0]);
32    //
33    // iterate
34    //
35    for ( int i=1 ; i<nmax ; i++ )
36    {
37      //
38      // some arithmetic
39      //
40      Precision n=i+1;
41      Precision sqrt_n=sqrt(n);
42      Precision sqrt_nm1=sqrt(n-1.0);
43      //
44      // recurence
45      //
46      Qn[i]=(s/sqrt_n)*(1.0-Pn[i])-sqrt_nm1/sqrt_n*Qn[i-1];
47    }
48    //
49    // return result
50    //
51    return Qn;
52  }
53
54  #endif
```

D. Function to compute $C'_n s$

```
1   /*! \file Compute_Cn.h
2       \brief Template function for computing Cn's values with desired precision
3   */
4
5   #ifndef COMPUTECN_H
6   #define COMPUTECN_H
7
8   #include <cmath>
9   #include <vector>
10  #include <iostream>
11
12  //! \brief recursive function for Cn(s) computation up to order n.
13  /// \param Pn Pn(s) values up to order n, without 1/(1+z) factor
14  /// \param t_over_tv real number, the fit parameter for Cns.
15  /// \param t_over_tphi real number, the fit parameter for Cns.
16  template <class Precision>
17  std::vector<Precision> compute_Cn(std::vector<Precision> &Pn, Precision t_over_tv, Precision t_over_tphi)
18  {
19    //
20    // get number of elements to be computed
21    //
22    int nmax=Pn.size();
23    //
24    // allocate memory
25    //
26    std::vector<Precision> Cn(nmax);
27    //
28    // 1/(1+z) factor
29    //
30    Precision factor=1.0/(1.0+t_over_tphi);
31    //
32    // loop on orders
33    //
34    for ( int i=0 ; i<nmax ; i++ )
35    {
36      Precision num_1=2.0*(1.0-t_over_tv)*(1.0-t_over_tv)*(1.0-t_over_tv)*Pn[i]*factor;
37
38      Precision num_2=Pn[i]*factor;
39
40      Precision num_3=(1.0-2.0*t_over_tv)*(1.0-2.0*t_over_tv)*(1.0-2.0*t_over_tv)*Pn[i]*factor;
41
42      Precision den_1=1.0-(1.0-t_over_tv)*Pn[i]*factor;
43
44      Precision den_2=1.0-Pn[i]*factor;
45
46      Precision den_3=1.0-(1.0-2.0*t_over_tv)*Pn[i]*factor;
47
48      Cn[i]=num_1/den_1+num_2/den_2-num_3/den_3;
49    }
```

```
  //
  // return result
  //
  return Cn;
}

#endif
```

E. Function to compute $\delta\sigma_{WL}$

```
/*! \file Compute_Sigma.h
    \brief Template function for computing Pn's values with desired precision
*/

#ifndef COMPUTESIGMA_H
#define COMPUTESIGMA_H

#include <cmath>
#include <vector>
#include <iostream>

//! \brief compute sigma_a contribution
///
/// compute b/pi * sum_n Cn*Pn
///
/// \param Pn Pn(s) values up to order n, without 1/(1+z) factor
/// \param Qn Qn(s) values up to order n, without 1/(1+z) factor
/// \param Cn Cn(s) values up to order n
/// \param b real number, the fit parameter for Cns.
/// \param t_over_tphi real number, the fit parameter for Cns.
template <class Precision>

Precision compute_sigma_a(std::vector<Precision> &Pn, std::vector<Precision> &Qn, std::vector<Precision> &Cn, Precision b, Precision t_over_tphi)

{
  //
  // get number of elements to be computed
  //
  int nmax=Pn.size();
  //
  // allocate memory
  //
  Precision sigma_a=0.0;
  //
  // loop on orders
  //
  for ( int i=0 ; i<nmax ; i++ )
  {
    sigma_a = sigma_a + Cn[i]*Pn[i]*Pn[i];
  }
  //
  // add factors
  //
  sigma_a = sigma_a*b/(1.0+t_over_tphi)/(1.0+t_over_tphi)/3.14159265358979323846264338327950288;
  //
  // return result
  //
  return sigma_a;
}

//! \brief compute sigma_b contribution
///
/// compute b/pi * sum_n (Cn+C_{n+1})*Qn/2
///
/// \param Pn Pn(s) values up to order n, without 1/(1+z) factor
/// \param Qn Qn(s) values up to order n, without 1/(1+z) factor
/// \param Cn Cn(s) values up to order n
/// \param b real number, the fit parameter for Cns.
/// \param t_over_tphi real number, the fit parameter for Cns.
template <class Precision>
Precision compute_sigma_b(std::vector<Precision> &Pn, std::vector<Precision> &Qn, std::vector<Precision> &Cn, Precision b, Precision t_over_tphi)
{
  //
  // get number of elements to be computed
  //
  int nmax=Pn.size();
  //
  // allocate memory
  //
  Precision sigma_b=0.0;
```

```
71    //
72    // loop on orders
73    //
74    for ( int i=0 ; i<nmax-1 ; i++ )
75    {
76      sigma_b = sigma_b + (Cn[i]+Cn[i+1])*Qn[i]*Qn[i];
77    }
78    //
79    // add factors
80    //
81    sigma_b = sigma_b*b/(1.0+t_over_tphi)/(1.0+t_over_tphi)/2.0/3.14159265358979323846264338327950 2;
82    //
83    // return result
84    //
85    return sigma_b;
86  }
87
88  #endif
```

---

[1] Kuntsevich, A.Y. *et al.* Intervalley scattering and weak localization in Si-based two-dimensional structures. *Phys. Rev. B.* **75**, 195330 (2007).

[2] Kawabata, A.Y On the field dependence of magnetoresistance in two-dimensional systems. *J. Phys. Soc. Jpn* **53**, 3540 (1984).

[3] McPhail, S. *et al.* Weak localization in high-quality two-dimensional systems. *Phys. Rev. B.* **70**, 245311 (2004).