Лекция №17. Расчеты

Сегодня поговорим про расчеты. Буду пороть правду-матку. Всемирные секреты всем и даром.

Расчеты бывают разные. Бьюсь об заклад, что большинству из вас до сих пор приходилось сталкиваться с расчетом чего-то весьма ограниченное количество раз. Как правило, для расчета дается какая-то формула или система формул (первая лабораторная), либо набор правил, которые затруднительно сформулировать в виде системы формул (вторая и третья лабораторые).

Для первого типа программирование довольно очевидно: как правило, используются какие-то итерационные формулы, реализуемые рекурсивно или в виде циклов. Какое-то время моей жизни я довольно много считал несобственных двойных и тройных интегралов. Примерно того вида, что был в конце прошлой лекции. Дам несколько подсказок:

1. методов интегрирования придумано великое множество и ограничиваться совсем одним, особенно если он требует много времени и не дает стабильного результата, не стоит;

2. Есть возможность найти нужную квадратуру (количество точек в интегрировании) под любое количество точек, но, как правило, в этом нет смысла — лучше разбить интервал интегрирования на большое количество кусков, каждый из которых интегрируется проверенной кубатурой. Я чаще всего пользовался методом Гаусса по 16 точкам. Точность достаточно высока, и существенно не растет, если увеличивать порядок. Под точностью я имею в виду не абстрактное интегрирование одного и того же интервала двумя разными методами, а точность при интегрировании одного большого набора отрезков при примерном равенстве времен расчета.

$$\int_a^b f(x)dx = \sum_i \int_{a_i}^{b_i} f(x)dx \tag{1}$$

3. Не нужно пользоваться для интегрирования методом прямоугольников.

4. Я сказал, **не нужно пользоваться для интегрирования методом прямоугольников**.

5. Используйте метод Монте-Карло для интегрирования только в крайних случаях, когда нужно на бегу написать программу для расчета

величины, жизненно вам необходимой с крайне невысокой точностью, но чтобы расчет завершился примерно в течение недели (это намек, что метод Монте-Карло использовать не стоит совсем).

6. Если у вас не крайне сложная функция, нет очень быстрых осцилляций, то спокойно пользуйтесь методом Гаусса, только надо коэффициенты в справочнике взять

7. Если у вас на расчет формула с особыми точками и быстро осциллирующими функциями, вы явно свернули не туда, зачем вам наука?

8. Но все же. Перед расчетом нужно внимательно рассмотреть подынтегральное выражение и проанализировать примерное его поведение. Найти особые точки и попытаться от них избавиться, но если не получается, придется пользоваться леммой Жордана и гнуть комплексный контур. Если вам придется такое делать, то это не сложно, но громоздко.

9. Судя по коду многих из вас, над оптимизацией вы не задумываетесь, это плохо. Нужно писать:

   - Коротко и понятно
   - С комментариями
   - Переменные не должны называться a,b,c,d, и не должны ThisIsVariableForXaxisCounterForProgramInnerLoop
   - По возможности, выделяйте функции
   - Функции должны быть не слишком длинными — так логика их работы легче анализируется и проверяется
   - ООП — штука совершенно прекрасная, но не нужно надеяться на автоматические системы. Высвобождайте память сами, удаляйте объекты, не пользуйтесь по возможности автоматическими структурами, работу которых не понимаете (я про вектор например)

10. Нет, я не против использования высокоуровневых языков и их сложных примитивов, но для расчетов? Для расчетов это не просто блажь, это легко может сделать секундный расчет часовым.

В конце приведу небольшой код из старых времен для расчета интегралов.

С задачей описания функции по ее диффиренциальному уравнению методом Рунге-Кутты вы уже столкнулись, так вот, по сути там использовались прямые численные методы. То есть вы знаете уравнение, знаете начальные условия — и пошли моделировать дискретизованную систему. Так

сейчас считается почти все, что угодно. Но есть один огромный недостаток — высокая требовательность к вычислительным ресурсам. Она действительно гигантская.

Если останется время и желание (у меня), постараюсь рассказать про FDTD (finite defined time dependence) — метод прямого численного моделирования распространения электромагнитных полей.

Второй тип задач. Которые сформулированы например как во второй лабораторной работе. Тут все просто и сложно одновременно. С одной стороны, это не формулы, тут не нужно думать, как их понять, какие методы использовать. С другой стороны, именно в таких задачах нужно составлять проект расчета до написания программы. Продумать полностью всю систему хранения данных, продумать последовательность выполнения операций, и только после этого садиться писать код. То есть по-хорошему, вы сначала пишете отчет по проекту, а только потом проект. Кроме предложенных лабораторных работ и еще пары-тройки клеточных автоматов и других дискретных распределенных систем, мне нечего привести в пример таких моделей. Про Ва-тор, Жизнь и ОДА я рассказывал достаточно подробно, в них есть, безусловно, общие части, но во многом они разнятся. Так что универсального метода расчета клеточных автоматов не существует. Метод придется придумывать при возникновении задачи.

С другой стороны, можно поучиться писать игры с нуля. Не с использованием движка. С нуля. Это и есть то самое математическое моделирование в чистом виде.

Контрольные вопросы

1. Как рассчитать / задать скорость объекта в дискретно определенной системе

2. Опишите вкратце метод интегрирования прямоугольниками и метод Монте-Карло

3. В чем основной недостаток прямых численных методов?

1 Приложение

```
module integrlib

  type limits
     double precision :: lower,upper
     integer :: qnt
  end type limits
  double precision,dimension(8) :: carr,warr
```

```
  data carr /&
       0.98940093499164992359_8, 0.94457502307323257607_8,&
       0.86563120238783174388_8, 0.75540440835500303389_8,&
       0.61787624440264374844_8, 0.45801677765722738634_8,&
       0.28160355077925891323_8, 0.09501250983763744018_8/
  data warr /&
       0.02715245941175409485_8, 0.06225352393864789286_8,&
       0.09515851168249278481_8, 0.12462897125553387205_8,&
       0.14959598881657673208_8, 0.16915651939500253818_8,&
       0.18260341504492358886_8, 0.18945061045506849628_8/
  integer :: tqnt = 2

contains

  subroutine uintgss(intlim1,intfnc,rsl)
    implicit none
    double precision :: plim1,stp1
    double precision :: rsl
    TYPE(limits) :: intlim1
    integer :: fcnt1,scnt1
    interface
       function intfnc(prm1) result(rsl)
         double precision :: prm1,rsl
       end function intfnc
    end interface
    stp1 = (intlim1%upper - intlim1%lower) / real(intlim1%qnt)
    rsl=0._8;
    do scnt1=1,intlim1%qnt;
       plim1=intlim1%lower+(scnt1-1)*stp1
       do fcnt1=1,8
          rsl=rsl+warr(fcnt1)*( &
               intfnc(&
               plim1+.5_8*stp1*(1._8+carr(fcnt1))&
               )+&
               intfnc(&
               plim1+.5_8*stp1*(1._8-carr(fcnt1))&
               )&
               )
       end do
    end do
    rsl=rsl*.5_8*stp1
  end subroutine uintgss

  subroutine dintgss(intlim1,intlim2,intfnc,rsl)
    implicit none
    double precision :: plim1,stp1,plim2,stp2
    double precision :: rsl
    TYPE(limits) :: intlim1,intlim2
    integer :: fcnt1,scnt1,fcnt2,scnt2
```

```
  interface
     function intfnc(arg1,arg2) result(rsl)
       double precision :: arg1,arg2,rsl
     end function intfnc
  end interface
  stp1 = (intlim1%upper - intlim1%lower) / real(intlim1%qnt)
  stp2 = (intlim2%upper - intlim2%lower) / real(intlim2%qnt)
  rsl=0._8;
  do scnt2=1,intlim2%qnt
     plim2=intlim2%lower+(scnt2-1)*stp2
     do scnt1=1,intlim1%qnt;
        plim1=intlim1%lower+(scnt1-1)*stp1
        do fcnt2=1,8
           do fcnt1=1,8
              rsl=rsl+warr(fcnt1)*warr(fcnt2)*( &
                   intfnc(&
                   plim1+.5_8*stp1*(1._8+carr(fcnt1)),&
                   plim2+.5_8*stp2*(1._8+carr(fcnt2))&
                   )+&
                   intfnc(&
                   plim1+.5_8*stp1*(1._8+carr(fcnt1)),&
                   plim2+.5_8*stp2*(1._8-carr(fcnt2))&
                   )+&
                   intfnc(&
                   plim1+.5_8*stp1*(1._8-carr(fcnt1)),&
                   plim2+.5_8*stp2*(1._8+carr(fcnt2))&
                   )+&
                   intfnc(&
                   plim1+.5_8*stp1*(1._8-carr(fcnt1)),&
                   plim2+.5_8*stp2*(1._8-carr(fcnt2))&
                   )&
                   )
           end do
        end do
     end do
  end do
  rsl=rsl*.25_8*stp1*stp2
end subroutine dintgss

subroutine uuint(intfnc, tablim, intlim1, tabprm)
  implicit none
  include 'mpif.h'
  TYPE(limits) ::  tablim, intlim1
  double precision, allocatable, dimension(:) :: farr, parr
  double precision :: tim1, tim2, tim3, tabprm, tabstp
  integer :: bcnt, ecnt, cnt, tcnt1, tcnt2!, tqnt
  integer :: MPI_size, MPI_rank, MPI_err, Status(MPI_status_size)

  interface
     function intfnc(prm1) result(rsl)
```

```
      double precision :: prm1,rsl
    end function intfnc
end interface

call MPI_Init(MPI_err)
call MPI_COMM_SIZE(MPI_COMM_WORLD, MPI_size, MPI_err)
call MPI_COMM_RANK(MPI_COMM_WORLD, MPI_rank, MPI_err)

bcnt = int(MPI_rank * tablim%qnt / real(MPI_size)) + 1
ecnt = int((MPI_rank + 1) * tablim%qnt / real(MPI_size))
tabstp = (tablim%upper - tablim%lower) / real(tablim%qnt)
if(MPI_rank.ne.0)then
   allocate(parr(bcnt : ecnt))

   do cnt=bcnt, ecnt
      tabprm = tablim%lower + tabstp * (cnt - .5_8)
      call uintgss(intlim1, intfnc, parr(cnt))
   end do

   call MPI_SEND(parr,ecnt - bcnt + 1, MPI_REAL8, 0, MPI_rank,&
        MPI_COMM_WORLD, MPI_err)
   deallocate(parr)
else
   tim1 = MPI_Wtime(); tim3 = tim1
   tcnt2 = 0;! tqnt = 10;

   allocate(farr(tablim%qnt))

   do cnt=1, ecnt
      tcnt1 = int(100 * cnt / real(ecnt * tqnt)); if(tcnt1 > tcnt2) then;
      tcnt2 = tcnt1; tim2= MPI_Wtime();
      write(0,'(i3,"% completed on ",f12.3," s. overall: ", f12.3, " s.")') &
           tcnt2 * tqnt, tim2 - tim3, tim2 - tim1; tim3 = tim2;  end if

      tabprm = tablim%lower + tabstp * (cnt - .5_8)
      call uintgss(intlim1, intfnc, farr(cnt))
   end do

   do cnt = 1, MPI_size - 1
      bcnt = int(cnt * tablim%qnt / real(MPI_size)) + 1;
      ecnt = int((cnt + 1) * tablim%qnt / real(MPI_size));
      call MPI_RECV(farr(bcnt : ecnt), ecnt - bcnt + 1, MPI_REAL8, cnt, cnt,&
           MPI_COMM_WORLD, Status, MPI_err)
   end do

   do cnt = 1, tablim%qnt
      tabprm = tablim%lower + tabstp * (cnt - .5_8)
      print*, tabprm, farr(cnt)
   end do
```

```
      deallocate(farr)
    end if

    call MPI_finalize(MPI_err);
  end subroutine uuint

  subroutine udint(intfnc, tablim, intlim1, intlim2, tabprm)
    implicit none
    include 'mpif.h'
    TYPE(limits) ::  tablim, intlim1, intlim2
    double precision, allocatable, dimension(:) :: farr, parr
    double precision :: tim1, tim2, tim3, tabprm, tabstp
    integer :: bcnt, ecnt, cnt, tcnt1, tcnt2!, tqnt
    integer :: MPI_size, MPI_rank, MPI_err, Status(MPI_status_size)

    interface
       function intfnc(prm1,prm2) result(rsl)
         double precision :: prm1,prm2,rsl
       end function intfnc
    end interface

    call MPI_Init(MPI_err)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, MPI_size, MPI_err)
    call MPI_COMM_RANK(MPI_COMM_WORLD, MPI_rank, MPI_err)

    bcnt = int(MPI_rank * tablim%qnt / real(MPI_size)) + 1
    ecnt = int((MPI_rank + 1) * tablim%qnt / real(MPI_size))
    tabstp = (tablim%upper - tablim%lower) / real(tablim%qnt)
    if(MPI_rank.ne.0)then
       allocate(parr(bcnt : ecnt))

       do cnt=bcnt, ecnt
          tabprm = tablim%lower + tabstp * (cnt - .5_8)
          call dintgss(intlim1, intlim2, intfnc, parr(cnt))
       end do

       call MPI_SEND(parr,ecnt - bcnt + 1, MPI_REAL8, 0, MPI_rank,&
            MPI_COMM_WORLD, MPI_err)
       deallocate(parr)
    else
       tim1 = MPI_Wtime(); tim3 = tim1
       tcnt2 = 0;! tqnt = 10;

       allocate(farr(tablim%qnt))

       do cnt=1, ecnt
          tcnt1 = int(100 * cnt / real(ecnt * tqnt)); if(tcnt1 > tcnt2) then;
          tcnt2 = tcnt1; tim2= MPI_Wtime();
          write(0,'(i3,"% completed on ",f12.3," s. overall: ", f12.3, " s.")') &
               tcnt2 * tqnt, tim2 - tim3, tim2 - tim1; tim3 = tim2;  end if
```

```
         tabprm = tablim%lower + tabstp * (cnt - .5_8)
         call dintgss(intlim1, intlim2, intfnc, farr(cnt))
      end do

      do cnt = 1, MPI_size - 1
         bcnt = int(cnt * tablim%qnt / real(MPI_size)) + 1;
         ecnt = int((cnt + 1) * tablim%qnt / real(MPI_size));
         call MPI_RECV(farr(bcnt : ecnt), ecnt - bcnt + 1, MPI_REAL8, cnt, cnt,&
              MPI_COMM_WORLD, Status, MPI_err)
      end do

      do cnt = 1, tablim%qnt
         tabprm = tablim%lower + tabstp * (cnt - .5_8)
         print*, tabprm, farr(cnt)
      end do

      deallocate(farr)
   end if

   call MPI_finalize(MPI_err);
end subroutine udint

subroutine duint(intfnc, tablim1, tablim2, intlim1, tabprm1, tabprm2)
   implicit none
   include 'mpif.h'
   TYPE(limits) ::  tablim1, tablim2, intlim1
   double precision, allocatable, dimension(:) :: farr, parr
   double precision :: tim1, tim2, tim3, tabprm1, tabprm2, tabstp1, tabstp2
   integer :: bcnt, ecnt, cnt, tcnt1, tcnt2
   integer :: MPI_size, MPI_rank, MPI_err, Status(MPI_status_size)

   interface
      function intfnc(arg1) result(rsl)
        double precision :: arg1,rsl
      end function intfnc
   end interface

   call MPI_Init(MPI_err)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, MPI_size, MPI_err)
   call MPI_COMM_RANK(MPI_COMM_WORLD, MPI_rank, MPI_err)

   bcnt = int(MPI_rank * tablim1%qnt * tablim2%qnt / real(MPI_size)) + 1
   ecnt = int((MPI_rank + 1) * tablim1%qnt * tablim2%qnt / real(MPI_size))
   tabstp1 = (tablim1%upper - tablim1%lower) / real(tablim1%qnt)
   tabstp2 = (tablim2%upper - tablim2%lower) / real(tablim2%qnt)
   if(MPI_rank.ne.0)then
      allocate(parr(bcnt : ecnt))

      do cnt=bcnt, ecnt
```

```
            tabprm1 = tablim1%lower + tabstp1 * (mod(cnt-1,tablim1%qnt) + .5_8)
            tabprm2 = tablim2%lower + tabstp2 * ((cnt-1)/tablim1%qnt + .5_8)
            call uintgss(intlim1, intfnc, parr(cnt))
         end do

         call MPI_SEND(parr,ecnt - bcnt + 1, MPI_REAL8, 0, MPI_rank,&
              MPI_COMM_WORLD, MPI_err)
         deallocate(parr)
      else
         tim1 = MPI_Wtime(); tim3 = tim1
         tcnt2 = 0;

         allocate(farr(tablim1%qnt*tablim2%qnt))

         do cnt=1, ecnt
            tcnt1 = int(100 * cnt / real(ecnt * tqnt)); if(tcnt1 > tcnt2) then;
            tcnt2 = tcnt1; tim2= MPI_Wtime();
            write(0,'(i3,"% completed on ",f12.3," s. overall: ", f12.3, " s.")') &
                 tcnt2 * tqnt, tim2 - tim3, tim2 - tim1; tim3 = tim2;  end if

            tabprm1 = tablim1%lower + tabstp1 * (mod(cnt-1,tablim1%qnt) + .5_8)
            tabprm2 = tablim2%lower + tabstp2 * ((cnt-1)/tablim1%qnt + .5_8)
            call uintgss(intlim1, intfnc, farr(cnt))
         end do

!        do cnt = 1, MPI_size - 1
         cnt = 1
         do while(cnt < MPI_size)
            bcnt = int(cnt * tablim1%qnt * tablim2%qnt / real(MPI_size)) + 1;
            ecnt = int((cnt + 1) * tablim1%qnt * tablim2%qnt / real(MPI_size));
            call MPI_RECV(farr(bcnt : ecnt), ecnt - bcnt + 1, MPI_REAL8, cnt, cnt,&
                 MPI_COMM_WORLD, Status, MPI_err)
            cnt = cnt + 1
         end do
!        end do

         do cnt = 1, tablim1%qnt*tablim2%qnt
            tabprm1 = tablim1%lower + tabstp1 * (mod(cnt-1,tablim1%qnt) + .5_8)
            tabprm2 = tablim2%lower + tabstp2 * ((cnt-1)/tablim1%qnt + .5_8)
            print*, tabprm1, tabprm2, farr(cnt)
            if(mod(cnt,tablim1%qnt).eq.0)print*
         end do

         deallocate(farr)
      end if

      call MPI_finalize(MPI_err);
    end subroutine duint

    subroutine ddint(intfnc, tablim1, tablim2, intlim1, intlim2, tabprm1, tabprm2)
```

```
implicit none
include 'mpif.h'
TYPE(limits) ::  tablim1, tablim2, intlim1, intlim2
double precision, allocatable, dimension(:) :: farr, parr
double precision :: tim1, tim2, tim3, tabprm1, tabprm2, tabstp1, tabstp2
integer :: bcnt, ecnt, cnt, tcnt1, tcnt2!, tqnt
integer :: MPI_size, MPI_rank, MPI_err, Status(MPI_status_size)

interface
   function intfnc(arg1,arg2) result(rsl)
     double precision :: arg1,arg2,rsl
   end function intfnc
end interface

call MPI_Init(MPI_err)
call MPI_COMM_SIZE(MPI_COMM_WORLD, MPI_size, MPI_err)
call MPI_COMM_RANK(MPI_COMM_WORLD, MPI_rank, MPI_err)

bcnt = int(MPI_rank * tablim1%qnt * tablim2%qnt / real(MPI_size)) + 1
ecnt = int((MPI_rank + 1) * tablim1%qnt * tablim1%qnt / real(MPI_size))
tabstp1 = (tablim1%upper - tablim1%lower) / real(tablim1%qnt)
tabstp2 = (tablim2%upper - tablim2%lower) / real(tablim2%qnt)
if(MPI_rank.ne.0)then
   allocate(parr(bcnt : ecnt))

   do cnt=bcnt, ecnt
      tabprm1 = tablim1%lower + tabstp1 * (mod(cnt-1,tablim1%qnt) + .5_8)
      tabprm2 = tablim2%lower + tabstp2 * ((cnt-1)/tablim1%qnt + .5_8)
      call dintgss(intlim1, intlim2, intfnc, parr(cnt))
   end do

   call MPI_SEND(parr,ecnt - bcnt + 1, MPI_REAL8, 0, MPI_rank,&
        MPI_COMM_WORLD, MPI_err)
   deallocate(parr)
else
   tim1 = MPI_Wtime(); tim3 = tim1
   tcnt2 = 0;

   allocate(farr(tablim1%qnt*tablim2%qnt))

   do cnt=1, ecnt
      tcnt1 = int(100 * cnt / real(ecnt * tqnt)); if(tcnt1 > tcnt2) then;
      tcnt2 = tcnt1; tim2= MPI_Wtime();
      write(0,'(i3,"% completed on ",f12.3," s. overall: ", f12.3, " s.")') &
           tcnt2 * tqnt, tim2 - tim3, tim2 - tim1; tim3 = tim2;  end if

      tabprm1 = tablim1%lower + tabstp1 * (mod(cnt-1,tablim1%qnt) + .5_8)
      tabprm2 = tablim2%lower + tabstp2 * ((cnt-1)/tablim1%qnt + .5_8)
      call dintgss(intlim1, intlim2, intfnc, farr(cnt))
   end do
```

```
      do cnt = 1, MPI_size - 1
         bcnt = int(cnt * tablim1%qnt * tablim2%qnt / real(MPI_size)) + 1;
         ecnt = int((cnt + 1) * tablim1%qnt * tablim2%qnt / real(MPI_size));
         call MPI_RECV(farr(bcnt : ecnt), ecnt - bcnt + 1, MPI_REAL8, cnt, cnt,&
              MPI_COMM_WORLD, Status, MPI_err)
      end do

      do cnt = 1, tablim1%qnt*tablim2%qnt
         tabprm1 = tablim1%lower + tabstp1 * (mod(cnt-1,tablim1%qnt) + .5_8)
         tabprm2 = tablim2%lower + tabstp2 * ((cnt-1)/tablim1%qnt + .5_8)
         print*, tabprm1, tabprm2, farr(cnt)
         if(mod(cnt,tablim1%qnt).eq.0)print*
      end do

      deallocate(farr)
    end if

    call MPI_finalize(MPI_err);
  end subroutine ddint

end module integrlib

module rayprop
  real(8) :: xsrc, zsrc, eps1, eps2, zbrd
contains

  subroutine optlng(x,z,delay,coef)
    implicit none
    real(8) :: x, z, delay, coef, sq, n, xrt, lng, sinp, sint, cosp, cost
    real(8) :: a,b,c,d,e,c1,c2
    sq(x)=x*x; n = sqrt(eps2 / eps1)
    if(z > zbrd) then
       xrt = xsrc + (x-xsrc)*(zsrc-zbrd)/(zsrc+z-2._16*zbrd)
       lng = sqrt(sq(x-xsrc)+sq(zsrc+z-2*zbrd))
       delay = lng*sqrt(eps1)
       sinp = (x-xsrc)/lng; sint = n*sinp
       cosp = sqrt(1-sq(sinp)); cost = sqrt(1-sq(sint))
       coef = (cosp/sqrt(eps2)-cost/sqrt(eps1))/(cosp/sqrt(eps2)+cost/sqrt(eps1))
    else
       c1 = (z-zbrd)*(z-zbrd)/(1._16-n*n)
       c2 = -n*n*(zsrc-zbrd)*(zsrc-zbrd)/(1._16-n*n)
       a = 1._16; b = -2._16*(x-xsrc)
       c = c1+c2+(x-xsrc)*(x-xsrc)
       d = -2._16*c2*(x-xsrc)
       e = c2*(x-xsrc)*(x-xsrc)
       xrt = point4(a,b,c,d,e) + xsrc
       lng = sqrt(sq(xrt-xsrc)+(zbrd-zsrc))
       sinp = (xrt-xsrc)/lng; sint = n*sinp
       delay = lng*sqrt(eps1)+sqrt(eps2*(sq(x-xrt)+sq(z-zbrd)))
```

```
      cosp = sqrt(1-sq(sinp)); cost = sqrt(1-sq(sint))
      coef = 2._16*cosp/sqrt(eps2)/(cosp/sqrt(eps2)+cost/sqrt(eps1))
    end if
  end subroutine optlng

  function point4(a,b,c,d,e) result(x)
    implicit none
    real(8) :: a,b,c,d,e
    real(8) :: alpha,beta,gamma
    complex(8) :: y,W,V,K,P,Q,R,U,x,x1,x2,x3,x4
    alpha = (-3._16*b*b/(8._16*a)+c)/a; beta = ((b*b/(4._16*a)-c)*b/(2*a)+d)/a
    gamma = (((-3._16*b*b/(16._16*a)+c)*b/(4._16*a)-d)*b/(4._16*a)+e)/a
    P = - alpha*alpha/12._16-gamma
    Q = - alpha*alpha*alpha/108._16+alpha*gamma/3._16-beta*beta/8._16
    R = - Q/2._16+sqrt(Q*Q/4._16+P*P*P/27._16)
    U = sqrt3(R); y = -5._16/6._16*alpha+U-P/(3._16*U)
    W = sqrt(alpha+2._16*y)
    x = - B/(4._16*A)+0.5_16*(W-sqrt(-(3*alpha+2*y+2*beta/W)))
  end function point4

  function sqrt3(in)result(out)
    implicit none
    complex(8) :: in,out
    out = exp(log(in)/3._16)
  end function sqrt3
end module rayprop
```