Numerical Integration in Cosmology

Anita Schael

March 30, 2006

1 Introduction

Numerical integration is a vital tool for carrying out cosmological computations. Assuming special cases for the cosmological constant it is possible to do integrals analytically, but in general this is not the case. We will have a look at an example where numerical integration is necessary to cover all likely possibilities of cosmological paramaters, and I will introduce different methods to solve this integral.

2 Cosmological Background

In cosmology the measurement of distances is an important prerequisite for obtaining further information about the structure of the early universe. Those measurements however are not trivial: as observers on Earth we are looking back in time when observing far-distant objects, and in particular the distances between objects are changing due to the expansion of the Universe.

There exist many different cosmological distance measures, but we will have a closer look at one in particular: the luminosity distance.

To give a quick overview of the relevant cosmological models we are using:

$$\Omega_m + \Omega_\Lambda + \Omega_k = 1$$

We are assuming a flat universe, this means $\Omega_k = 0$, therefore $\Omega_m + \Omega_\Lambda = 1$; furthermore

$$\Omega_m = 0.3, \quad \Omega_\Lambda = 0.7 \qquad \text{and} \quad H_o = 70\ km\ s^{-1} Mpc^{-1}. \tag{1}$$

Luminosity distance

The Luminosity distance D_L is related to the flux measured by an observer by

$$F_o = \frac{L}{4\pi D_L^2} \tag{2}$$

D_L is also related to other cosmological distances by

$$D_L = (1+z)D_M = (1+z)^2 D_A \tag{3}$$

where D_M and D_A are called the transverse comoving distance, and the angular diameter distances, respectively (see Hogg 2000 for details). The transverse comoving distance D_M is actually equivalent to the proper motion distance, which is defined as transverse velocity over proper motion. D_M is equal to the effective distance D_{eff} in the case of a flat universe $\Omega_k = 0$, which is the case we are looking at.
That means:

$$D_L = (1+z)D_{eff} \tag{4}$$

D_{eff} can be derived from the distance-redshift relation:

$$Rdr = \frac{c}{H(z)}dz \tag{5}$$

Integrate:

$$rR = c\int_o^z \frac{dz}{H(z)} \tag{6}$$

For $\Omega_k = 0$ (the flat model) $rR = D_{eff}$, and using equation (4) we obtain:

$$D_L = (1+z)\ c\int_o^z \frac{dz}{H(z)} \tag{7}$$

We can define the function $H(z)$ as:

$$H(z) = H_o\left((1+z)^2(1+\Omega_m z) - z(2+z)\Omega_\Lambda\right)^{1/2} \tag{8}$$

(see Carroll, Press, Turner 1992 for detailed derivation), so that our final equation looks like follows:

$$D_L = (1+z)\frac{c}{H_o}\int_o^z \frac{dz}{((1+z)^2(1+\Omega_m z) - z(2+z)\Omega_\Lambda)^{1/2}} \tag{9}$$

This integral can be solved analytically for the special case $\Omega_\Lambda = 0$ and $\Omega_m = 1$ (Weinberg 1972, Kolb and Turner 1990), but in our case, see (1), we will have to use numerical integration.

3 Numerical Integration

Now we will have look at the actual numerical integration process to solve equation (9). The general idea of integrating a function which cannot be done analytically is to split it up into equally spaced steps whose values are known. It is, of course, the aim to obtain the integral as accurately as possible with as few steps of function evaluations of the integrand as possible.

There are methods of various orders, however higher order does not necessarily mean higher accuracy - although in several cases it does. (See Press: Numerical Recipes in Fortran for further details about this section.)

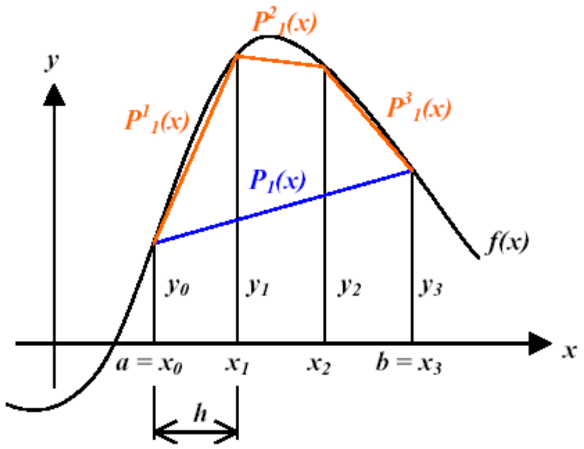

Figure 1: Function with equally spaced intervals h in the x-coordinate to visualise the way the trapezoidal rule works (from: Meiler, M.: Numerische Methoden der praktischen Mathematik, 2005)

3.1 General Methods

Simple formulas

The very basic idea of approximating the integral of a given function is expressed in the *Trapezoidal rule*:

$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] + O(h^3 f'') \tag{10}$$

The value of the function will in this case be simply estimated by drawing a straight line between the limits a and b and calculating the average (see figure 1). It is obvious that, in most simple cases, the smaller h, the better the estimate.

The error term $O(h^3 f'')$ is the amount by which our approach differs from the true solution. In this case this is proportional to the value of the function's second derivative somewhere in the interval of integration.

We see immediately that this equation is exact for any polynomial up to degree 1, for example $f(x) = 2x$, since in this case the second derivative equals zero in all points. The trapezoidal rule (10) is called a two-point formula (x_1 and x_2) for obvious reasons.

Since many formulas we use unfortunately exceed degree 1, we will have to find a slightly more complex method to evaluate equations of higher degrees. The next step will therefore be to have a look at a three-point formula, which is already accurate for polynomials up to and including degree 3, for example $f(x) = 2x^3$. This equation is called *Simpson's rule*:

$$\int_{x_1}^{x_3} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] + O(h^5 f^{(4)}) \tag{11}$$

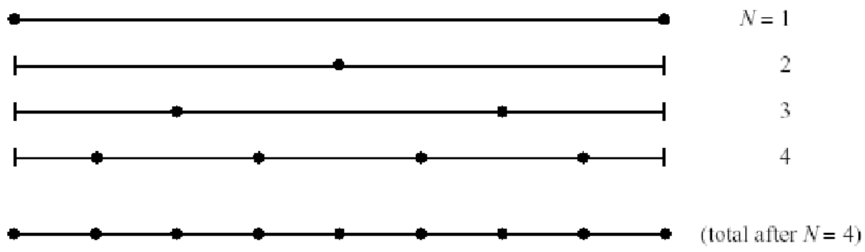

Figure 2: Different stages of refinement for the routine `trapzd`. Information from previous calls will be maintained, and newly computed steps of refinement are simply added (from: Press, W.H.: Numerical Recipes in Fortran)

There are, of course, more formulas for even higher degrees, but for our case the Simpson's rule up to degree 3 will be more than sufficient.

Extended formulas

To actually use those formulas we will have to extend them to carry out the basic computation n times (rather: $n-1$ times). For the example of the simple trapezoidal rule we use equation (10) $n-1$ times by integrating the intervals $(x_1, x_2), (x_2, x_3), ..., (x_{n-1}, x_n)$, and just add the results. The *Extended trapezoidal rule* therefore looks as follows:

$$\int_{x_1}^{x_n} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + f_3 + \cdots + f_{n-1} + \frac{1}{2}f_n\right] + O\left(\frac{(b-a)^3 f''}{n^2}\right) \tag{12}$$

Equally we obtain the *Extended Simpson's rule*:

$$\int_{x_1}^{x_n} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \frac{4}{3}f_4 + \cdots + \frac{2}{3}f_{n-2} + \frac{4}{3}f_{n-1} + \frac{1}{3}f_n\right] + O\left(\frac{1}{n^4}\right) \tag{13}$$

Based on those equations we can now start to write routines that will carry out the computations for us.

3.2 Algorithms

First we will develop a piece of code performing the extended trapezoidal rule. The extended trapezoidal rule is an important starting point for several algorithms, mainly due to the fact that we can double the number of intervals without losing the benefit of previous work. With every stage of refinement we simply add the value of the function at the halfway points of the previous endpoints (see figure 2).

It can be seen that the first and coarsest step is simply to average the function at its endpoints a and b. The next step of refinement will be to add the value of the function at the halfway point, then at the 1/4 and 3/4 points and so on.

A routine carrying out exactly that computation will look as follows:

```
SUBROUTINE trapzd(func,a,b,s,n)
INTEGER n
REAL a,b,s,func
EXTERNAL func
    This routine computes the nth stage of refinement of an extended trapezoidal rule. func is
    input as the name of the function to be integrated between limits a and b, also input. When
    called with n=1, the routine returns as s the crudest estimate of ∫_a^b f(x)dx. Subsequent
    calls with n=2,3,... (in that sequential order) will improve the accuracy of s by adding 2^(n-2)
    additional interior points. s should not be modified between sequential calls.
INTEGER it,j
REAL del,sum,tnm,x
if (n.eq.1) then
    s=0.5*(b-a)*(func(a)+func(b))
else
    it=2**(n-2)
    tnm=it
    del=(b-a)/tnm                  This is the spacing of the points to be added.
    x=a+0.5*del
    sum=0.
    do 11 j=1,it
        sum=sum+func(x)
        x=x+del
    enddo 11
    s=0.5*(s+(b-a)*sum/tnm)        This replaces s by its refined value.
endif
return
END
```

This is the basic routine for integrating a function by the extended trapezoidal rule. Here the number of steps, n, is still unknown. In principle it would be possible to determine n in advance and then just carry out the computation n times. This is obviously not very practical, so we need another routine to specify n: we refine the calculation more and more until a certain degree of accuracy is achieved. The routine carrying out these steps is called `qtrap`.

`qtrap` provides a good way to evaluate functions and is fairly sufficient for not very smooth functions, e.g. a function of a variable that is linearly interpolated between measured data points. In fact, a higher-order method would not increase the accuracy in this case since those methods are only more efficient for rather smooth integrands.

For function of a higher degree, however, we can take advantage of the Simpson's rule (see (13)) and create a subroutine called `qsimp`. A routine performing the integration by the Simpson's rule is generally more efficient than `qtrap` when the integrated function has a continuous 3rd derivative. Note that the routine `qsimp` will be just used in place of `qtrap`. Both routines need their workhorse `trapzd` to perform the actual computation.

```
      SUBROUTINE qtrap(func,a,b,s)
      INTEGER JMAX
      REAL a,b,func,s,EPS
      EXTERNAL func
      PARAMETER (EPS=1.e-6, JMAX=20)
C     USES trapzd
         Returns as s the integral of the function func from a to b. The parameters EPS can be set
         to the desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum
         allowed number of steps. Integration is performed by the trapezoidal rule.
      INTEGER j
      REAL olds
      olds=-1.e30                  Any number that is unlikely to be the average of the function
      do 11 j=1,JMAX                  at its endpoints will do here.
          call trapzd(func,a,b,s,j)
          if (j.gt.5) then         Avoid spurious early convergence.
              if (abs(s-olds).lt.EPS*abs(olds).or.
     *                (s.eq.0..and.olds.eq.0.)) return
          endif
          olds=s
      enddo 11
      pause 'too many steps in qtrap'
      END
```

```
      SUBROUTINE qsimp(func,a,b,s)
      INTEGER JMAX
      REAL a,b,func,s,EPS
      EXTERNAL func
      PARAMETER (EPS=1.e-6, JMAX=20)
C     USES trapzd
         Returns as s the integral of the function func from a to b. The parameters EPS can be set
         to the desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum
         allowed number of steps. Integration is performed by Simpson's rule.
      INTEGER j
      REAL os,ost,st
      ost=-1.e30
      os= -1.e30
      do 11 j=1,JMAX
          call trapzd(func,a,b,st,j)
          s=(4.*st-ost)/3.              Compare equation (4.2.4), above.
          if (j.gt.5) then              Avoid spurious early convergence.
              if (abs(s-os).lt.EPS*abs(os).or.
     *                (s.eq.0..and.os.eq.0.)) return
          endif
          os=s
          ost=st
      enddo 11
      pause 'too many steps in qsimp'
      END
```

4 Application

In this section we want to finally have a look at how to apply the given routines to our problem in equation (9). We are now equipped with the necessary tools and simply have to adjust the routines to compute our primary function: the luminosity distance D_L.
Our final equation looked like:

$$D_L = (1+z)\frac{c}{H_o}\int_o^z \frac{dz}{((1+z)^2(1+\Omega_m z) - z(2+z)\Omega_\Lambda)^{1/2}} \quad (14)$$

where the term in the denominator proved to be the analytically unsolvable function. We will therefore determine a little function called `func` which contains exactly that term and will be the function to be integrated by `trapzd`.

```
real*8 function func(om_l,om_m,z)
implicit none
real*8 om_l, om_m, z

func = ((1.0d0+z)**2*(1.0d0+om_m*z)-z*(2.0d0+z)*om_l)**(-0.5d0)

end function
```

This function will be read in by the performing routine - I have chosen to use `qsimp`, although `qtrap` would have been equally sufficient in this case.

```
      subroutine qsimp(func,a,b,s,om_m, om_l)
      implicit none
      integer jmax, i
      real*8 a,b,func,s,eps, os,ost,st, om_m, om_l
      external func
      parameter (eps=1.e-6, jmax=20)

c     Uses trapzd and returns as s the integral of the function func
c     from a to b. The parameters eps can be set to the desired
c     fractional accuracy and jmax so that 2 to the power jmax-1 is the
c     maximum allowed number of steps. Integration is performed by
c     Simpson's rule.

      ost = -1.e30
      os = -1.e30

      do i=1,jmax
            call trapzd(func,a,b,st,i,om_m,om_l)
            s = (4.*st-ost)/3.

            if (i.gt.5) then
                  if (abs(s-os).lt.eps*abs(os).or.
     +            (s.eq.0..and.os.eq.0.)) return
            end if

            os = s
            ost = st
      end do
      pause 'too many steps in qsimp'

      end
```

The core calculation routine, `trapzd`, needed to be only slightly adjusted and extended by the necessary parameters used in `func`.

Finally, the actual calculation of the luminosity distance D_L will be carried out by the subroutine `lumdist` which contains equation (14) and uses in place of the integral the value returned by `qsimp`.

```fortran
      subroutine trapzd(func,a,b,s,n,om_m,om_l)
      implicit none
      integer n, i, j
      real*8 a,b,s,func,jr,sum,space,x2,om_l,om_m
      external func

c     This routine computes the nth stage of refinement of an extended
c     trapezoidal rule. func is input as the name of the function to be
c     integrated between limits a and b, also input.

      if (n.eq.1) then
            s = 0.5*(b-a)*(func(om_l,om_m,a)+func(om_l,om_m,b))
      else

c     Number of point to be added
            j = 2**(n-2)
            jr = j
c     Spacing of points to be added
            space = (b-a)/jr
            x2 = a + 0.5*space
            sum = 0.

            do i=1,j
                  sum = sum+func(om_l,om_m,x2)
                  x2 = x2+space
            end do

            s = 0.5*(s + (b-a)*sum/jr)
      end if
      return

      end
```

```fortran
      subroutine lumdist(D_L, z, om_l, om_m, c, s, func, x2, H_o)

      implicit none
      real*8 D_L, z, om_l, om_m, c, s, func, x2, H_o
      external func

      x2 = 0.

c     Call qsimp to calculate the integral by using Simpson's rule
      call qsimp(func,x2,z,s,om_m,om_l)

c     Do the final calculation for the luminosity distance D_L

      D_L = (1.0d0+z)*(c/H_o)*s

      end
```

This is the subroutine which will be called in the overall programming code to provide us immediately with the luminosity distance D_L at our given redshift z.

I hope very much this seminar was interesting and useful for some of you, and that you now, should you have to begin to work with numerical integration, will have a bit more of an idea about how to get started.

References

[1] Carroll, S.M., Press, W.H., Turner, E.L., 1992, The cosmological constant, ARA&A 30 499-542

[2] Hogg, D.W., 2000, Distance measures in Cosmology

[3] Kolb, E.W., Turner, M.S., 1990, The Early Universe, Addison-Wesley, Redwood City

[4] Meiler, M., 2005, Numerische Methoden der praktischen Mathematik, Institut für Informatik, Universität Leipzig

[5] Press, W.H., 2001, Numerical Recipes in Fortran: The Art of Scientific Computing, Vol. 1, Cambridge University Press

[6] Weinberg, S., 1972, Gravitaion and Cosmology: Principles and Applications of the General Theory of Relativity, John Wiley & Sons, New York