CALCULATING DIRECTIVITIES WITH THE 2D SIMPSON'S RULE

J. C. Brégains[1] (Student Member, IEEE), I. C. Coleman,[2]

F. Ares[1] (Senior Member, IEEE) and E. Moreno[1] (Senior Member, IEEE)

[1] Grupo de Sistemas Radiantes, Departamento de Física Aplicada, Facultad de Física,
Universidad de Santiago de Compostela.
15782 - Santiago de Compostela – Spain

[2]Servicio de Traducción, Universidad de Santiago de Compostela.
15782 - Santiago de Compostela – Spain

e-mail (by order of author): fajulio@usc.es, stdirec@usc.es, faares@usc.es, famoreno@usc.es

ABSTRACT

For calculating antenna directivities, C++ routines implementing the simple generalization of Simpson's rule to two dimensions compare well with methods currently available in common symbolic and numerical calculation packages.

1. INTRODUCTION

The directivity $D(\theta_0,\phi_0)$ of an antenna in the direction (θ_0,ϕ_0) is given by

$$D_0 = \frac{\text{Power in the direction } (\theta_0,\phi_0)}{\text{Average power}} = \frac{4\pi\left|\xi(\theta_0,\phi_0)\right|^2}{\int_0^{2\pi}\int_0^{\pi}\left|\xi(\theta,\phi)\right|^2 \sin\theta \, d\theta \, d\phi} \tag{1}$$

where $|\xi(\theta,\phi)|^2$ is the power radiated in the direction (θ,ϕ).[1] When the integration in the denominator has to be performed numerically, calculation of the directivity is generally time-consuming in comparison with other calculations that are commonly required in antenna synthesis. When repeated directivity calculations are required, as in optimization-based array synthesis procedures that take directivity into account, they can constitute the rate-limiting step of the overall computational process. Although the computational burden can often be alleviated by applying symmetry considerations (for example, for arrays of elements that are isotropic or have a common element factor that can be ignored, for which the integral reduces to a linear combination of the element excitations [1]; or for line antennas and others with rotational symmetry, for which the double integral reduces to a single integral), in many cases this is not possible – notably when

[1] For planar antennas the range of integration in the denominator is limited to the upper hemisphere, $(0,0) \leq (\theta,\phi) \leq (\pi/2,2\pi)$.

$\xi(\theta,\phi)$ is known only through a set of field measurements. Approximations are available that can be applied in certain situations [2-4], but they are only approximations, and are by no means universally applicable; when they are not applicable, it is likely that most antenna designers nowadays resort to the integration routines of widely used mathematics packages such as MATLAB [5], Mathematica [6] and IMSL [7], or incorporate well-known numerical integration algorithms [8] in *ad hoc* programs written in languages such as C or C++. In this paper we present a C++ routine for directivity calculation that is based on the generalization of Simpson's rule to two dimensions, and show that this and an alternative C++ 2D Simpson method both compare well with other approaches [8]. As far as the authors know, this method has not, surprisingly (and probably due to its ease), been discussed before in the technical literature, despite some rough publications in certain web pages.

2. THE TWO-DIMENSIONAL SIMPSON'S RULE

Let us first recall the one-dimensional Simpson's rule. To approximate a definite integral $I = \int_a^b f(x)\,dx$, we first decompose it as a sum of integrals over smaller equal intervals,

$$I = \sum_{n=0}^{N-1} I_n\,, \quad I_n = \int_{x_n}^{x_{n+1}} f(x)\,dx \tag{2}$$

where $x_0 = a$ and $x_N = b$, and then apply a quadrature formula to each I_n, i.e. we approximate I_n as a linear combination $I_n = \sum_{k=0}^{P} w_k\, f(x_{n,k})$ of the values of f at certain points $x_{n,k}$ in $[x_n,x_{n+1}]$. Simpson's rule is a quadrature formula that uses three points ($x_{n,0} = x_n$, $x_{n,1} = x_n + ½\Delta = ½(x_n+x_{n+1})$, and $x_{n,2} = x_n + \Delta = x_{n+1}$, where $\Delta = x_{n+1} - x_n$) and requires that the approximation actually be exact for polynomials of order 0, 1 and 2. When applied, in particular, to the polynomials $f(x) = 1$, $f(x) = x$ and $f(x) = x^2$, this requirement leads to the system of equations

$$\begin{array}{ccccccccc} 1w_0 & + & 1w_1 & + & 1w_2 & = & \int_{x_n}^{x_{n+1}} 1 dx & = & \Delta \\ w_0\,x_n & + & \tfrac{1}{2} w_1 (x_n + x_{n+1}) & + & w_2\,x_{n+1} & = & \int_{x_n}^{x_{n+1}} x\,dx & = & \tfrac{1}{2}(x_n + x_{n+1})\Delta \\ w_0\,x_n^2 & + & \tfrac{1}{4} w_1 (x_n + x_{n+1})^2 & + & w_2\,x_{n+1}^2 & = & \int_{x_n}^{x_{n+1}} x^2\,dx & = & \tfrac{1}{3}(x_n^2 + x_n x_{n+1} + x_{n+1}^2)\Delta \end{array} \tag{3}$$

which can be solved to afford $w_0 = w_2 = \Delta/6$ and $w_1 = 4\Delta/6$. Thus:

$$I \approx (\Delta/6) \sum_{n=0}^{N-1} \sum_{k=0}^{2} q_k\, f(x_{n,k}) \tag{4}$$

where $\Delta = (b–a)/N$ and $\{q_0,q_1,q_2\} = \{1,4,1\}$, with an error that can be shown to be of the order of Δ^4. Finally, taking into account that $x_{n,2} = x_{n+1,0} = x_{n+1}$, eq.4 becomes

$$I \approx (\Delta/6)\sum_{i=0}^{2N} Q_i f(u_i) \tag{5}$$

where $u_i = x_0 + i\Delta/2$ (i = 0,...,2N) and the Q_i follow the pattern 1,4,2,4,2,....,2,4,2,4,1, i.e. $Q_0 = Q_{2N} = 1$, $Q_i = 2$ if i is even ($i \neq 0,2N$), $Q_i = 4$ if i is odd.

The generalization of the above derivation to integrals over rectangular areas in two dimensions is straightforward. To approximate a definite integral $J = \int_a^b \int_c^d f(x,y)\,dy\,dx$, we first decompose it as a sum over equal rectangles $[x_n,x_{n+1}]\times[y_m,y_{m+1}]$,

$$J = \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} J_{nm}, \quad J_{nm} = \int_{x_n}^{x_{n+1}}\int_{y_m}^{y_{m+1}} f(x,y)\,dy\,dx \tag{6}$$

where $x_0 = a$, $x_N = b$, $y_0 = c$ and $y_M = d$; and we then apply to each J_{nm} a (3 × 3)-point quadrature formula in which the points are defined in each direction as for the one-dimensional Simpson's rule, and which we require to be exact for all monomials $f(x,y) = x^r y^s$ (r,s = 0,1,2), This requirement leads to a system of nine linear equations of the form

$$\sum_{k=0}^{2}\sum_{l=0}^{2} w_{k,l}\,(x_{n,k})^r\,(y_{m,l})^s = \left(\frac{x_{n+1}^{r+1} - x_n^{r+1}}{r+1}\right)\left(\frac{y_{m+1}^{s+1} - y_m^{s+1}}{s+1}\right) \tag{7}$$

that can be solved to yield the solution:

$$w_{k,l} = (1/36)\Delta_x\Delta_y q_{k,l} \tag{8}$$

where $\Delta_x = x_{n+1} - x_n = (b-a)/N,\ \Delta_y = y_{m+1} - y_m = (d-c)/M$ and the $q_{k,l}$ are the elements of the matrix

$$\begin{bmatrix} 1 & 4 & 1 \\ 4 & 16 & 4 \\ 1 & 4 & 1 \end{bmatrix}$$

Finally, then

$$J \approx (1/36)\Delta_x\Delta_y \sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\sum_{k=0}^{2}\sum_{l=0}^{2} q_{k,l}\, f(x_{n,k}, y_{m,l}) \tag{9}$$

with an error that can be shown to be of the order of $\Delta_x^4 + \Delta_y^4$; and when it is taken into account that $x_{n,2} = x_{n+1,0} = x_{n+1}$ and $y_{m,2} = y_{m+1,0} = y_{m+1}$, equation 9 becomes

$$J \approx (1/36)\Delta_x\Delta_y \sum_{i=0}^{2N}\sum_{j=0}^{2M} Q_{i,j}\, f(u_i, v_j) \tag{10}$$

where $u_i=x_0+i\Delta_x/2$, $v_j=y_0+j\Delta_y/2$ and the $Q_{i,j}$ follow the one–dimensional Simpson pattern along the boundaries of the integration domain (i. e. if i is 0 or $2N$, or j is 0 or $2M$) and in the interior of the domain $Q_{i,j}=Q_{0,j}Q_{i,0}$, as in the following scheme:

Q_{ij}	i=0	i=1	i=2	i=3	·	·	·	i=2N-3	i=2N-2	i=2N-1	i=2N
j=0	1	4	2	4	·	·	·	4	2	4	1
j=1	4	16	8	16	·	·	·	16	8	16	4
j=2	2	8	4	8	·	·	·	8	4	8	2
j=3	4	16	8	16	·	·	·	16	8	16	4
·	·	·	·	·	·	·	·	·	·	·	·
·	·	·	·	·	·	·	·	·	·	·	·
·	·	·	·	·	·	·	·	·	·	·	·
j=2M-3	4	16	8	16	·	·	·	16	8	16	4
j=2M-2	2	8	4	8	·	·	·	8	4	8	2
j=2M-1	4	16	8	16	·	·	·	16	8	16	4
j=2M	1	4	2	4	·	·	·	4	2	4	1

The C++ routine `dintsimpit` listed in the Appendix integrates a function pointed to by `funcp` over the rectangular domain [`ix,fx`]×[`iy,fy`] by repeated application of eq.10 (with $M = N$) to progressively finer subdivisions of the domain, halting when a convergence criterion is satisfied or a user-specified maximum number of iterations have been performed.

3. APPLICATION AND COMPARISON WITH OTHER METHODS

To test `dintsimpit` we compared its performance with those of some other commonly used two-dimensional integration tools, namely the MATLAB routine `dblquad` [5], the Mathematica routine `NIntegrate` [6], the IMSL Fortran routine `TWODQ` [7], and `dqsimp`, the nested application (for two-dimensional integration) of the C++ version of the one-dimensional algorithm `qsimp` of *Numerical Recipes* [8]. We used each method to calculate the denominator of equation 1 for a linear array of 10 isotropic elements spaced $\lambda/2$ apart on the z axis, for which [2]

$$\xi(\theta,\phi) = \frac{\sin(5\pi\cos\theta)}{\sin\left(\frac{\pi}{2}\cos\theta\right)} \tag{11}$$

Although the symmetry of this array leads to the desired integral being easily calculable as $10\times4\pi = 125.6637$, it nevertheless serves just as well as any other antenna for initial comparison of the efficiencies of the various two-dimensional integration routines. In running `NIntegrate`, an accuracy of three decimal places was required (`AccuracyGoal` = 3), and run times were measured using the function `Timing` [6]; `dblquad` was used with the error tolerance parameter set to 10^{-3}, and timed using the functions `tic` and `toc` [5]; and the other routines were terminated when the difference between the value of the integral obtained in successive iterations was less than 10^{-3}, and were timed using *ad hoc* timing routines. In all cases, $\xi(\theta,\phi)$ was calculated directly using equation 11, without any attempt to convert this expression to a computationally optimal form. All calculations were performed on a PC with an Athlon XP 1800 processor running at 1.533 GHz.

Table 1 lists the time taken by each method to calculate the desired integral 200 times. As expected, the MATLAB routine is by far the slowest (in spite of not achieving the desired accuracy) because it is run *via* an interpreter. The fact that the professional package routines `NIntegrate` and `TWODQ` are slightly slower than the C++ routines may be attributed to the former having been designed for integration of general functions over domains that are not necessarily rectangular, so that they presumably employ methods that are not necessarily the most efficient for integrating an antenna power pattern over a rectangle.

ROUTINE	**TIME (S)**	**RESULT**
dqsimp (C++)	0.75	125.664
dintsimpit (C++)	1.27	125.664
TWODQ (IMSL Fortran)	1.45	125.664
NIntegrate (Mathematica)	2.21	125.664
dblquad (MATLAB)	13.11	125.667

Table 1. Times taken by various routines to calculate the denominator of equation 1 two hundred times for a 10-element linear array antenna.

As a more exhaustive test, the two C++ Simpson routines were also used to calculate, with an absolute precision of 0.1, the directivities at $(\pi/4,\pi/4)$ of a series of planar arrays of $n_x \times n_y$ elements ($n_y = n_x + 5$), each with element factor $f_e(\theta,\phi) = \cos^2\theta$, that are laid out on rectangular grids with between-element spacing of $\lambda/2$ in both the x and y directions, the excitation I_{mn} of the (m,n)-th element being I_mI_n, where I_m and I_n are respectively the excitations of the m-th and n-th elements of n_x- and n_y-element linear arrays generating 25 dB Chebyshev patterns with their main beams aimed by introduction of a phase shift of $\pi/4$ between successive elements. A routine to do this using `dintsimpit` is listed in section (*b*) of the Appendix. Fig.1 shows that for arrays of more than about 2000 elements `dintsimpit` systematically performed better than `dqsimp`; although the cause of this is not totally clear, it seems likely to be due to the precision required of the intermediate one-dimensional integrations performed by `dqsimp` not being properly matched to the global precision requirement. The routine `dintsimpit` of course uses much more memory than `dqsimp`, but this is not likely to be a problem with today's computers.

4. CONCLUSIONS AND FINAL REMARKS

The calculation of directivities can become the rate-limiting step in antenna design procedures. The C++ integration routines tested in this study, both of which implement the two-dimensional Simpson's rule, appears to be for directivity calculation, at least, as fast as the general-purpose routines included in the commercial packages IMSL, Mathematica and MATLAB.

In practice, the repeated calculation of directivities in an antenna design optimization procedure can be speeded up even more by by-passing the progressive subdivision of the domain of integration that is normally required in an integration routine to ensure that a sufficiently good approximation to the desired integral is obtained. The routine `dintsimpit` (or `dqsimp`) can be used to calculate directivity at a single arbitrary point of the search space, and the number of divisions with which this calculation converges can then be forced during the whole of the optimization proper by setting `nbdivmin` appropriately and `niter` to 1, which considerably reduces computational overheads. For example, when the calculation for which results are shown in Table 1 was performed using `dintsimpit` with `niter` = 1 and `nbdivmin` = 21, the time taken for the 200 repetitions was 0.31 s. Once a near-optimum solution has been obtained in this way, the solution can be refined using the full resources of `dintsimpit`.

ACKNOWLEDGEMENTS

This work was supported by the Spanish Ministry of Science and Technology under project TIC 2002-04084-C03-02.

REFERENCES

[1] Y. U. Kim, “Peak Directivity Optimization under Side Lobe Level Constraints in Antenna Arrays”, *Electromagnetics*, vol. 8, no.1, pp. 51-70, 1988.

[2] C. A. Balanis, *Antenna Theory. Analysis and Design*, 2nd Edition, John Wiley and Sons, Inc., New York, 1997.

[3] R. S. Elliott, *Antenna Theory and Design*, Revised Edition, IEEE-Press, Wiley-Interscience, 2003.

[4] Chen-To Tai, and C. S. Pereira, “An Approximate Formula for Calculating the Directivity of an Antenna”, *IEEE Trans. on Antennas and Propagat.*, vol. 24, pp. 235 - 236, 1976.

[5] The MATLAB Group, Inc., *MATLAB Functions: Volume 1 (A-E)*, pdf document available through the MATLAB Help menu, 2002.

[6] S. Wolfram, *The Mathematica Book*, 4th Edition, Cambridge University Press, 1999.

[7] Visual Numerics, Inc., *IMSL Fortran Subroutines for Mathematical Applications, Vol. 1*, pdf document available through the Help menu of the Pro Fortran 8.2 Absoft Developer Tools Interface, 1997.

[8] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, 2nd Edition, Cambridge University Press, 1992.

APPENDIX

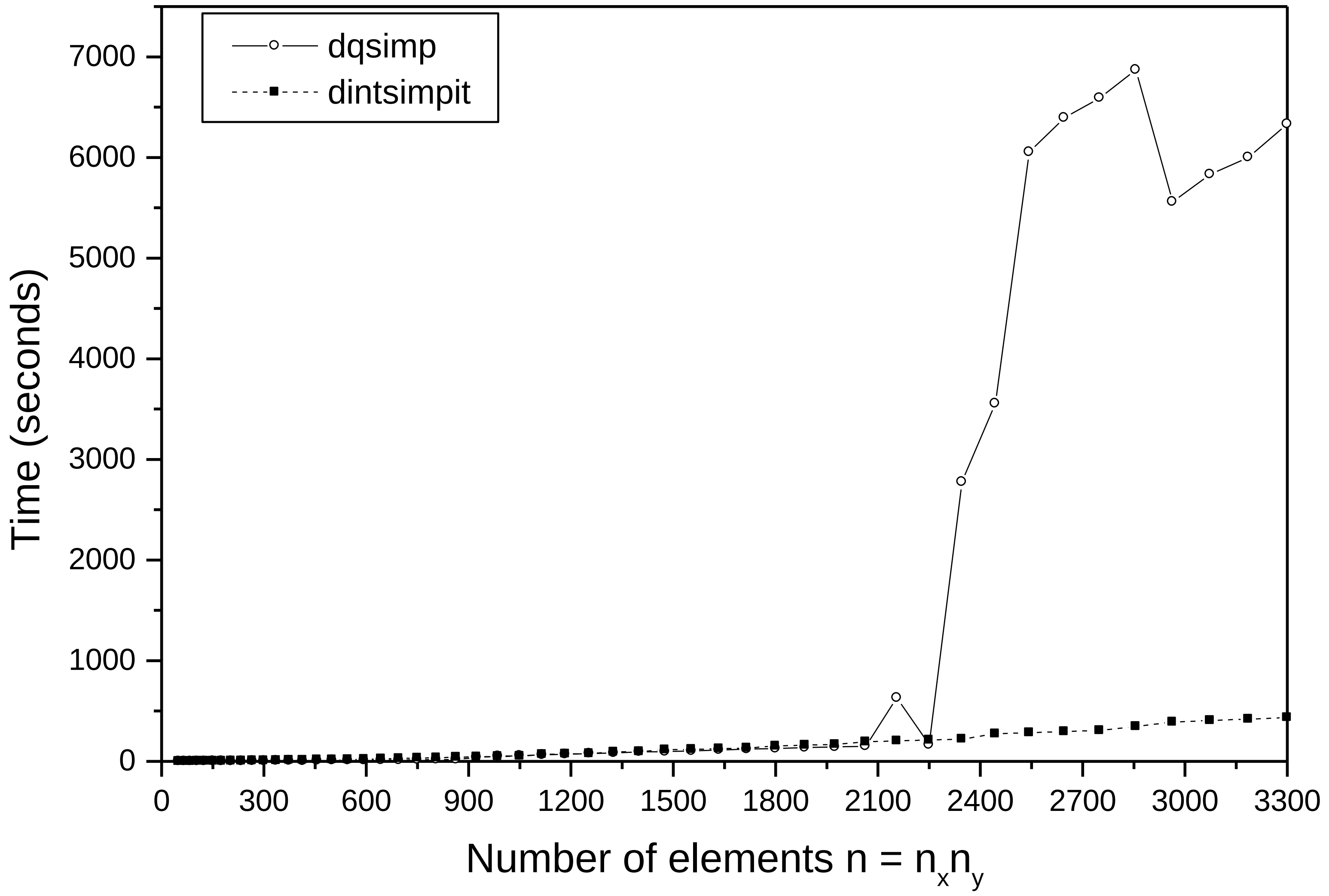
a) An annotated listing of the 2D Simpson's rule integration routine `dintsimpit`.

```
double dintsimpit(double (*funcp)(double,double), double ix, double fx,
            double iy, double fy,int nbdivmin , int niter , double precision)
{
/************************************************************************
 funcp = pointer to the function to be integrated (the integrand)
 x,y = arguments of the integrand
 ix,iy,fx,fy = initial and final values defining the domain of integration
 niter = max. no. of iterations (step-halvings) that algorithm will attempt
 precision = required absolute precision
 nbdivmin = number of "big" divisions of the domain of integration
            (N in eq.10) in the first iteration
 nsdivmin = number of "small" divisions of the domain of integration
            (2N in eq.10) in the first iteration
 nsdivmax = number of "small" divisions in the niter-th iteration
 relscale = current number of small divisions divided by nsdivmin
 n_points_max = maximum number of points in each dimension
 grid_size = n_points_max * n_points_max)
 minstep_x, minstep_y = size of smallest "small" divisions in each dimension
 dx, dy = size of current "small" divisions in each dimension
 f_val = vector storing calculated values of the integrand
 flags = boolean vector flagging points at which integrand has been calculated
 index = place in the vectors f_val and flags
 edge_x, edge_y = flags indicating points on the boundary of the domain
 old_s, new_s = old and new values of the integral
 sum = partial sum accumulated in each iteration
 a = vector of constants used for speed
 i,j,k,m,n = counters
 coeff = coefficient by which the value of the integrand at the current point
         is multiplied (see scheme in Section 2 above)
************************************************************************/
int index,i,j,k,m,n,relscale =(int)pow(2,niter-1),nsdivmin=2*nbdivmin,
nsdivmax=nbdivmin*(int)pow(2,niter);
int n_points_max=nsdivmax+1, grid_size=n_points_max*n_points_max;
double coeff, sum, old_s, new_s, *f_val, a[]={4,2};
double x,y,dx,dy, minstep_x=(fx-ix)/(double)nsdivmax,
      minstep_y=(fy-iy)/(double)nsdivmax;
bool *flags, edge_x, edge_y;
f_val=new double[grid_size]; flags=new bool[grid_size];
for(i=0;i<grid_size;i++) { f_val[i]=0; flags[i]=0;}
      for(i=0;i<niter;i++)
      {
            // dx, dy and relscale cannot be zero:

            if(relscale==0) relscale=1;
            dx=minstep_x*(double)relscale;
            dy=minstep_y*(double)relscale;

            // Initialize sum:

            sum=0;
            for(j=0,x=ix,m=0;j<n_points_max;j+=relscale,m++,x+=dx)
            {
                  edge_x=(j==0 || j==nsdivmax);
                  for(k=0,y=iy,n=0;k<n_points_max;k+=relscale,n++,y+=dy)
                  {
                        edge_y=(k==0 || k==nsdivmax);
                        index=j+k*n_points_max;

                  // If the point is a vertex, the coefficient is 1:
                    if(edge_x && edge_y) coeff=1;

                // Otherwise, the value of coeff depends on whether the point
                // lies on an edge and on the parities of m and n:

                    else
                    coeff=(((double)(m%2)+1)*((double)(n%2)+1))*
                        a[(int)(edge_x || edge_y)];

                // If the integrand has not already been calculated
                // for the current point, calculate it now:

                    if(flags[index]==0)
                    {
                        f_val[index]=funcp(x,y);
                        flags[index]=1;
                    }

                    // Now add the current term to the sum:

                    sum+=coeff*f_val[index];
                }
            }

            // In the first iteration, just set old_s and new_s
            // and reduce relscale:

            if(i==0) { old_s=new_s=sum*dx*dy/9.00; relscale/=2; continue; }

            // Otherwise, update old_s and new_s ...

            else { old_s=new_s; new_s=sum*dx*dy/9.00; }

            // ... test the halt criterion ...

            if(fabs(old_s-new_s)<=precision)
            {
                delete[] f_val,flags;
                return new_s;
            }

            // ... and reduce relscale if the desired precision
            // has not yet been achieved

            relscale/=2;
        }

        // If at this point we are still in dintsimpit, we clear memory; ...

        delete[] f_val,flags;

        // ... if we have not forced a single-iteration calculation
        // we announce that the algorithm has failed to converge; ...

        if(niter!=1)
        {
        cout << "Subroutine dintsimpit has not converged to the required";
        cout << " precision within the allowed number of iterations." << endl;
        }

        //... and we return the best value we have been able to obtain:

        return new_s;
}
```

b) A routine that uses `dintsimpit` to calculate the directivity of a planar antenna in the direction ($\pi/4,\pi/4$). The power radiated in any direction (`th`,`ph`) is given by the function `power`, and the integrand in the denominator of equation 1 is provided by the function `integrand`. The call to `dintsimpit` specifies that the denominator of equation 1 be calculated with a precision of 0.001 in at most 3 iterations, the domain of integration having initially been divided into 11 "big" divisions in each direction ($N = M = 11$ in equation 10).

```
#include <math.h>
#include <iostream.h>
#define PI 3.1415926535897932
double power(double theta, double phi);
double integrand(double theta, double phi);
double dintsimpit(double (*funcp)(double,double), double ix, double fx,
      double iy, double fy,int nbdivmin , int niter , double precision);
int main(void)
{
      double th=PI/4, ph=PI/4;
      double denom=dintsimpit(integrand,0,PI/2,0,2*PI,11,3,1E-3);
      if(denom<=0)
      {
            cout << "Problems with denominator. Abnormal termination" << endl;
            return 0;
      }
      cout << "Directivity =" << 4*PI*power(th,ph)/denom << endl;
      return 0;
}
double power(double theta, double phi)
{
      // CODE FOR power TO BE INSERTED HERE
}
double integrand(double theta, double phi)
{
      return power(theta,phi)*sin(theta);
}
double dintsimpit(double (*funcp)(double,double), double ix, double fx,
      double iy, double fy,int nbdivmin , int niter , double precision)
{
      // CODE SHOWN IN SECTION (a) OF THIS APPENDIX TO BE INSERTED HERE
}
```

Figure 1. Times taken to calculate the directivities of $\{n_x \times (n_x + 5)\}$-element arrays in a given direction using `dintsimpit` and `dqsimp`.


dqsimp
dintsimpit
Time (seconds)
Number of elements n = n_x n_y