

The Tortoise *and* the Hare

InterCall connects *Mathematica* to externally compiled code, such as the extensive body of numerical routines in the NAG, IMSL, and netlib libraries. Built atop *MathLink*, it makes foreign Fortran, C or Pascal procedures appear like *Mathematica* functions, and it can interpret a wider class of *Mathematica* structures than the built-in `Compile` function. InterCall represents the most significant advance in augmenting *Mathematica*'s numerical reach since the introduction of *MathLink*.

Sha Xin Wei

InterCall 2.1 (\$500; \$300 educational), from Analytica, P.O. Box 522, Nedlands, WA 6009, Australia. Fax/Phone: 61 9 386 5666. E-mail: analytic@earwax.pd.uwa.edu.au. InterCall works under *Mathematica* Versions 2.x and requires either a UNIX kernel or a Macintosh running MacTCP. InterCall is distributed by e-mail, *ftp*, and on 3.5 inch disk in *tar* or Macintosh format.

The Schrödinger wave equation:

$$\Delta P + i \partial_t P = VP$$

is an example of a fundamental problem where *Mathematica* hits the wall. We would like to solve this complex second order, parabolic partial differential equation for some boundary conditions in the plane. In cases where numerical methods have been developed, it's useful to seek help outside *Mathematica* rather than reinvent the wheel, poorly. InterCall connects *Mathematica* with the universe of numerical packages, and vastly extends *Mathematica*'s utility as a high-level symbolic synthesizer of large numerical analysis routines. InterCall's author, Terry Robb, has published several notebooks demonstrating its utility, including one showing off several quantum mechanical experiments solving the two-dimensional Schrödinger wave equation.

InterCall is based on Wolfram Research's *MathLink* external process communications protocol, and in fact may be viewed simplistically as a wrapper on *MathLink* optimized for passing real or complex scalar and array data to Fortran or C functions.

Typically, the numerical libraries reside on a host computer to which you would connect from *Mathematica*. Of course, the host computer may be the same as the computer on which you run *Mathematica*. You or the system administrator must first compile and install the library of executable functions on the host computer. To use InterCall, you load the package into your local *Mathematica* session, use Inter-

Call to connect to the host computer, and read in an InterCall database of definitions of the functions you wish to call from the external library on the host computer. To use a particular function from a library, you must install it before you can call it.

As an example, let us find the eigensystem of a tridiagonal matrix:

```
In[1]:= a = Table[ Switch[i-j, 1, -1, 0, 2, -1, -1, _, 0],
                {i, 1, 20}, {j, 1, 20} ];
```

```
In[2]:= << InterCall.m
        Loading InterCall version 2.1.
        Copyright (c) 1992-93 T. D. Robb.
```

Once InterCall is loaded, we connect to the host machine:

```
In[3]:= InterCall["elaine20.Stanford.EDU"];
```

Next, we load the InterCall database:

```
In[4]:= << InterData.m
```

and install a particular function from the external library:

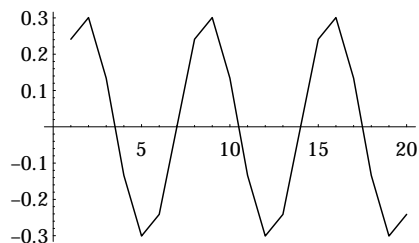
```
In[5]:= GetDefault[f02abf]
Out[5]= F02ABF[ (* TYPE=S *)
           $A -> In,          (* DATA=R[$IA, $N] *)
           $IA -> $N,        (* DATA=I *)
           $N -> COLS[$A],   (* DATA=I *)
           $R -> Out,        (* DATA=R[$N] *)
           $V // Transpose -> Out, (* DATA=R[$IV, $N] *)
           $IV -> $N,        (* DATA=I *)
           $E -> Null,       (* DATA=R[$N] *)
           $IFAIL -> -1      (* DATA=I *)
           ] (* CODE="LIBRARY" *)
f02abf[$A_] -> {$R, Crypto$V}
```

The last line of this output gives the calling sequence for f02abf. The next input invokes the function:

```
In[6]:= {eigvals, eigvecs} = f02abf[a];
Out[6]= InterCall::opened: Opened connection to host
eLaine20.Stanford.EDU
InterCall::import: Importing: {F02ABF}
InterCall::linked: Using remote driver version 2.0 on host
eLaine20.Stanford.EDU
```

Just to check, we plot the sixth eigenvector:

```
In[7]:= ListPlot[eigvecs[[6]] , PlotJoined -> True]
```



A function that is accessible to InterCall is not loaded into memory until it's actually invoked or loaded using InterCall's Import function. A point of possible confusion is that, although the package is loaded into the (local) kernel, the functions it calls may be executed in an entirely different process, perhaps on a different computer. This is quite different from *Mathematica's* import mechanism, which requires an imported function to cohabit with the kernel.

Why use InterCall?

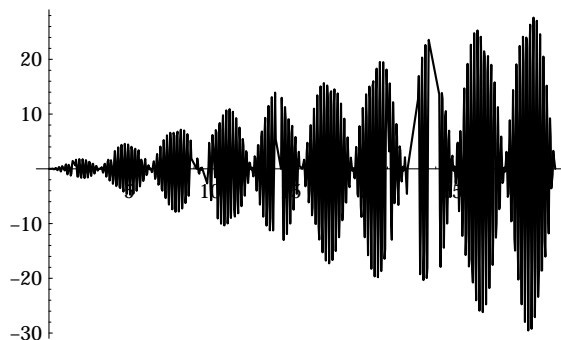
Arguably the most valuable enhancement to *Mathematica* offered by InterCall is speed. Using compiled Fortran functions confers an incredible speedup over equivalent built-in *Mathematica* methods: You can expect to gain an order of magnitude on sufficiently large problems. For small numeric problems requiring a single, straightforward scalar routine, *Mathematica's* built-in import mechanisms suffice, while InterCall would be overkill. But InterCall can now compile a larger class of functions than *Mathematica* itself.

What constitutes a "small" problem depends very much on your system and network. For example, the IMSL library of numerical routines is installed in a remote location on my network, so it takes a significant amount of time to communicate with those remote routines. In the following example, *Mathematica* took 22 wall-clock seconds to integrate

```
In[8]:= f[x_] := x Cos[30x] Sin[x]
```

over the interval $[0, 10\pi]$:

```
In[9]:= Plot[f[x], {x, 0, 10 Pi}]
```



```
In[10]:= sess = SessionTime[];
In[11]:= NIntegrate[f[x], {x, 0, 10 Pi}]
NIntegrate::slwcon:
Numerical integration converging too slowly; suspect
one of the following: singularity, oscillatory
integrand, or insufficient WorkingPrecision.
NIntegrate::slwcon:
Numerical integration converging too slowly; suspect
one of the following: singularity, oscillatory
integrand, or insufficient WorkingPrecision.
NIntegrate::slwcon:
Numerical integration converging too slowly; suspect
one of the following: singularity, oscillatory
integrand, or insufficient WorkingPrecision.
General::stop:
Further output of NIntegrate::slwcon
will be suppressed during this calculation.
NIntegrate::ncvb:
NIntegrate failed to converge to prescribed accuracy
after 7 recursive bisections in x near x = 29.3297.
```

```
Out[11]= 0.0349454
```

```
In[12]:= SessionTime[] - sess
```

```
Out[12]= 22.064791
```

The same computation took 213 seconds using the IMSL *dqdag* routine called via InterCall:

```
In[13]:= GetDefault[dqdag]
Out[13]= DQDAG [ (* TYPE=S *)
          $F -> In, (* DATA=RF[R] *)
          $A -> In, (* DATA=R *)
          $B -> In, (* DATA=R *)
          $ERRABS -> 0., (* DATA=R *)
          $ERRREL -> 0.00001, (* DATA=R *)
          $IRULE -> 1, (* DATA=I *)
          $RESULT -> Out, (* DATA=R *)
          $ERREST -> dqdag`errest (* DATA=R *)
          ] (* CODE="LIBRARY" *)
dqdag[$F_, $A_, $B_] -> $RESULT
```

```

In[14]:= Import[dqdag]
sess = SessionTime[];
dqdag[Function[x, x Cos[30 x] Sin[x]], 0, 10 Pi]

InterCall::opened:

  Opened connection to host elaine20.Stanford.EDU
InterCall::import: Importing: {DQDAG}
InterCall::linked:

  Using remote driver version 2.0 on host
  elaine20.Stanford.EDU

Out[14]= 0.034945413276557

In[15]:= SessionTime[] - sess
Out[15]= 213.952037

```

Most of this time is spent dynamically importing the required code, which is done only once, when you first invoke the external library function. To eliminate this problem, InterCall also provides a special compiler named *icc* so you can prepare code in advance, at the cost of some disk space.

InterCall converts *Mathematica* data types into the types expected by the libraries. It assumes that real data is double, not single precision. It converts between regular *Mathematica* tensors and external numeric arrays, setting the dimensions and types of the arrays as necessary, which saves a great deal of tedium.

InterCall provides two ways to introduce your own numeric functions into *Mathematica*: You can define a function within *Mathematica* and then have InterCall compile it, or you can write it in a low-level language like C or Fortran, compile it with a system tool like *cc*, and then import it using InterCall. The first approach is more convenient since the syntax and output are very much like *Mathematica's* `Compile`, but the second yields the fastest computations.

One of the most exciting promises of InterCall may be its independent compiler, which can already handle a superset of the functions *Mathematica's* built-in compiler can understand. For example, it can now compile array-functions, and when it is complete it should be able to deal with list functions like `Transpose`, dear to *Mathematica* (and APL) programmers. Fortunately, InterCall's author chose to adopt the pseudo-code set used by *Mathematica's* compiler.

InterCall's `find` function provides a simple and useful way to hunt for an algorithm in a library. For example:

```

In[16]:= find[ "integrate" && "IMSL" ]
Out[16]= DQAND (IMSL)
  Integrate a function on a hyper-rectangle.
DQDAG (IMSL)
  Integrate a function using a globally adaptive scheme
  based on Gauss-Kronrod rules.
DQDAGI (IMSL)
  Integrate a function over an infinite or semi-infinite
  interval.
DQDAGP (IMSL)
  Integrate a function with singularity points given.

```

```

DQDAGS (IMSL)
  Integrate a function (which may have endpoint
  singularities).
DQDAWC (IMSL)
  Integrate a function F(x)/(x-c) in the Cauchy principal
  value sense.
DQDAWO (IMSL)
  Integrate a function containing a sine or a cosine.
DQDAWS (IMSL)
  Integrate a function with algebraic-logarithmic
  singularities.
DQDNG (IMSL)
  Integrate a smooth function using a nonadaptive rule.

```

The `find` function recognizes a few simple Boolean patterns and operators like *Mathematica's* built-in string patterns. (Of course, one can fool such non-semantic searches using patterns like

```
find[ "integrate" && "IMSL" && Not["finite"]]
```

which will omit routines described by the term “infinite.”)

InterCall provides very fine control of two-way communication between *Mathematica* and your external code, using facilities to inspect and update data structures in the foreign code from within *Mathematica*. You can even import an external function compiled by InterCall and apply it as you would an `InterpolatingFunction`.

Names and Syntax

InterCall's only awkwardnesses lie in syntax. *Mathematica's* designers have been careful to name its functions so that we can guess their nature from the names alone, without reading obscure documentation. *Mathematica* functions are often defined with a clear syntax so that they behave as expected with typical arguments. Unfortunately, almost every other numerical analysis library and symbolic algebra system hosts a zoo of cryptic names, auxiliary parameters, and byzantine argument conventions. Connecting *Mathematica* to external libraries like NAG and IMSL necessarily exposes it to the obscure name space of these libraries. To help streamline usage, InterCall provides an extensive set of defaults for many of the parameters.

Even a simple application illustrates the problem. In one example in the Samples chapter of the manual (#5, page 30), the problem is to minimize:

```
In[17]= f[x_, y_, z_] := (x-1)^2 + (y-2)^2 + (z-3)^2;
```

In *Mathematica*, the call would be:

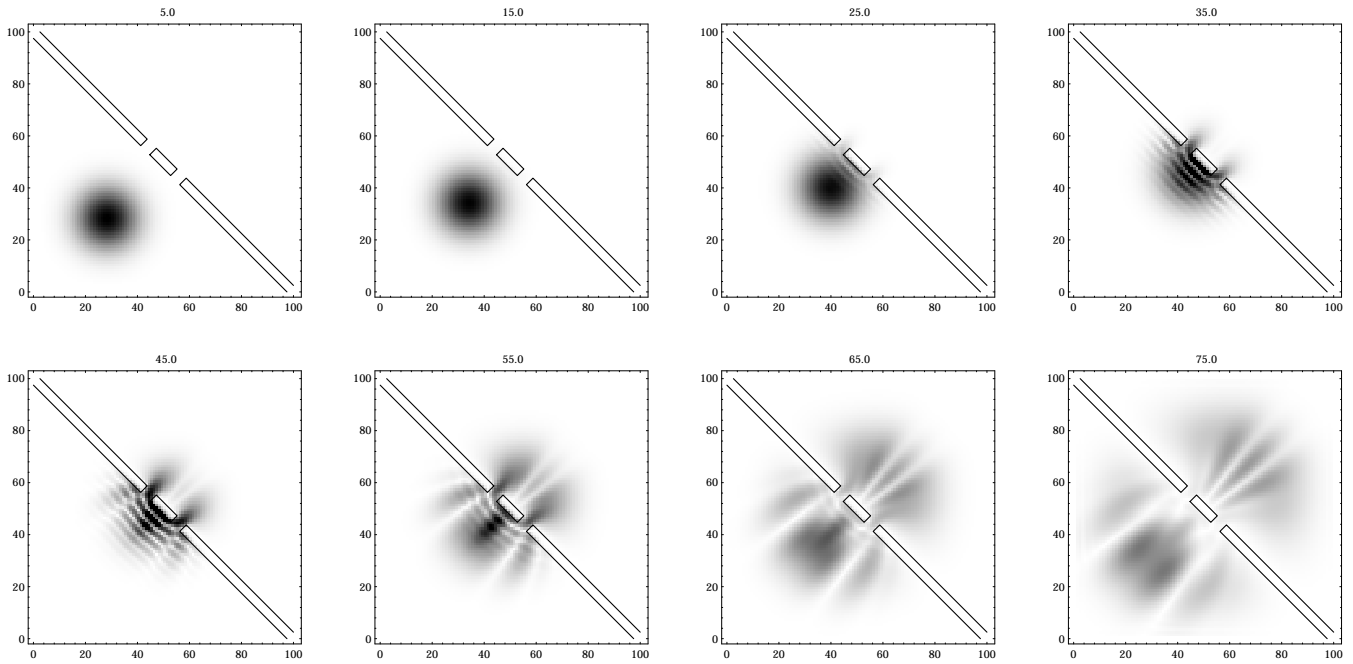
```
In[18]= FindMinimum[f[x, y, z], {x, 1}, {y, 1}, {z, 1}]
FindMinimum::fmgz:
```

```

Warning: FindMinimum encountered a vanishing gradient.
The result returned may not be a minimum; it may be a
maximum or a saddle point.

```

```
Out[18]= {0., {x -> 1., y -> 2., z -> 3.}}
```



According to the manual, the same problem is solved as follows using InterCall. The function that minimizes a multivariate function is `e04dgrf`:

```
In[19]:= GetDefault[e04dgrf]
```

Skipping some lines of output, we get:

```
Out[19]= e04dgrf[{$OBJFUN_, $X_} -> {$OBJF, $X}
```

Now we define our function:

```
In[20]:= myfunct = Function[{$mode, $n, $xc, $fc, $gc},
Module[{x, y, z},
{x, y, z} = $xc;
$fc = (x-1)^2 + (y-2)^2 + (z-3)^2;
$gc = 2*{x-1, y-2, z-3};] ];
```

```
In[21]:= e04dgrf[myfunct, {1, 1, 1}]
```

```
Out[21]= {0., {1., 2., 3.}}
```

Some extra work is required to call the external minimization routine as one would call *Mathematica*'s, without explicitly supplying the partial derivatives:

```
In[22]:= myfunct =.;
```

```
In[23]:= SetDefault[ e04dgrf,
{$OBJFUN -> ( Function[{$mode,$n,$xc,$fc,$gc},
$fc = In@@$xc;
$gc = Through[GC@@$xc];] /.
GC -> Table[Partial[$i][In], {$i, 1, $N}] ) ]
```

```
Out[23]= e04dgrf[Pseudo$OBJFUN_, $X_] -> {$OBJF, $X}
```

```
In[24]:= e04dgrf[ Function[{x, y, z},
(x-1)^2 + (y-2)^2 + (z-3)^2], {1, 1, 1} ]
```

```
Out[24]= {0., {1., 2., 3.}}
```

Installation, Support, and Documentation

InterCall is distributed by e-mail, *ftp*, and on disk in *tar* or Macintosh format. The UNIX version is thoughtfully packaged for automatic extraction using a standard shell command. I installed my examination copy on a NeXT running *Mathematica 2.2*, and was able to execute code on the same machine and remotely on several Sparcs. Drivers to run code on remote computers can be obtained for Apollo, Connection Machine, DEC/Ultix, HP 9000, IBM RISC, Iris, Sparc, Sun-3, VAX/VMS, and Cray Y-MP. InterCall shares *Mathematica*'s license protection policy. I found the distributor to be quite responsive and helpful.

InterCall's error messages are reasonably informative. However, the usage statements for the core InterCall functions seem too terse to be much use and the *Mathematica* help operator provides information on a function only after it has been loaded by `GetDefault`, `AddDefault`, or `Import`.

The manual (in TeX) is admirably extensive, but it could be more clearly divided between tutorial and reference sections. The examples come from Fortran, so they may be less than enlightening to researchers who are not familiar with the traditional culture of numerical analysis.

Conclusion

InterCall extends a programmer's reach tremendously far into whole categories of computational problems formerly closed to direct *Mathematica* manipulation. As an example, consider the two-dimensional Schrödinger wave equation introduced at the beginning of this review. Using InterCall and the external Schrödinger code, we can insert various barriers, scatter particles, and view the results "live" in a quantum-mechanical microworld. The figure above shows some frames from an animation created by Terry Robb of the two-slit interference experiment.

For those who can afford the resources, InterCall is arguably one of the best ways to unify symbolic with numerical analysis. ☞