

Remote Kernels, ComplexExpand, and Memory Use under Windows

Members of the Wolfram Research technical-support staff present articles on problems and questions we often encounter. In this column, we provide tips on running remote kernels from a Macintosh, using the `ComplexExpand` function to extend symbolic formulae to the complex numbers and setting the kernel memory allocation on a Windows machine.

Edited by Todd Gayley

Running Remote Kernels from a Macintosh

Kenneth J. Paradise
kj@wri.com

Macintosh users with network or dial-up access to a UNIX workstation running *Mathematica* can have transparent access to the computational power of the workstation without changing the way they work. Running remote kernels from a Macintosh is very easy, but users encounter few common mistakes and problems.

Beginning with Version 2.2, all communication between the front end and kernel uses the *MathLink* protocol. (The older, less capable *MathTalk* protocol is still supported, and it is required for connecting to Version 2.1 or earlier kernels.) Establishing a *MathLink* connection conceptually is very simple: The front end opens a link in Listen mode, and then the kernel opens a link in Connect mode, specifying where to “find” the front end’s side of the connection. All the complexity in configuring a remote kernel derives from the front end’s ability to automate this process. It is therefore instructive, and useful for troubleshooting, to perform these steps manually.

Start up *Mathematica* on your Macintosh, pull down the **Action** menu, and select **Kernels and Tasks**. A dialog box will pop up. Click on the **New Kernel** button and the **Kernel Configuration** dialog will appear. Select **Advanced Options** and change the “Arguments to MLOpen” to read:

```
-LinkMode Listen -LinkProtocol TCP
```

Delete the contents, if any, of the “Communication Toolbox Login” box. Click **OK** to return to the **Kernels and Tasks** dialog, then click the **Launch Kernel** button. An information box will come up with something like “Listening on 3041@kjmac.wri.com.” Your number and machine name will be different, but for my example I will use these. Now, open a connection to your UNIX machine. This can be done with any terminal program, or you can walk over to the machine yourself. When you get a UNIX prompt, type the following:

```
math -mathLink -linkmode connect -linkname 3041@kjmac.wri.com
```

Again, you will have a different number and machine name, but the other parts of the command will be the same. You will now have established the connection. You can go back to the front end and try a few calculations to make certain.

Several things might go wrong with the process just described. First, you need to have MacTCP installed on your Macintosh. MacTCP is included in the *Mathematica 2.2* distribution on disk 1. If you are using System 7.1 or later, you need MacTCP 1.1.1 or later. This is the version of MacTCP included with *Mathematica 2.2*. I have had complaints that MacTCP 2.02 does not work with *Mathematica*, but I have had no problems with it. Furthermore, if MacTCP is not configured correctly or if your UNIX machine does not recognize your Macintosh as a trusted host (that is, your Mac is not listed in the `/etc/hosts` file on the UNIX machine), then you may run into problems.

To automate the process of launching the remote kernel, the front end needs to perform the following steps: Open a telnet session with the UNIX machine, log in, and launch *Mathematica* with the proper command-line options. The key to this process is the “Remote Login” box in the **Kernel Configuration** dialog. What goes in this box is a simple script that will complete the login process and whatever other steps are necessary to get you to the UNIX shell prompt. The default script looks like

```
^:username\r^:password\r^>
```

The `^` means “wait for the following character,” so this script can be read as “wait for a colon, send the user name string followed by a carriage return, wait for another colon, send the password string followed by a carriage return, then wait for a `>` character.” This default script may not be appropriate for your system. For example, if your UNIX prompt does not end in a `>`, you will need to change the login string to reflect your prompt (for example, `%` and `$` are common prompt characters). When the last “wait for” in the login string has been satisfied, the front end will send the command to start the kernel. This command is a long string that begins with “`math -mathLink...`” Immediately after sending this string, the telnet window will disappear. In Version 2.1 and earlier, the

login string was optional, and you could type the command to launch the kernel yourself. In Version 2.2, the front end must send this command itself, so having a correct login string is necessary. If the `math` command line is sent too early, or not at all, you know your login string is not correct for the sequence of steps needed to get to a shell prompt on your system. I suggest you use a separate telnet program to log into the machine, taking careful note of the exact sequence of events required. You do not have to put all your inputs into the login string (for example, you may want to leave out your password) because you can type them in manually during the login, but the “wait for” commands must be correct in number and kind.

Sometimes the connection fails even though the `math` command string is sent at the right time. If that happens, there is a trick to prevent the telnet window from disappearing right away so you can see the exact command string that was sent and also any error messages that may have been returned. In the **Kernel Configuration** dialog, select the **Advanced Options** radio button. The “Communications Toolbox Login” box shows the complete login script, including the command that will be sent to launch *Mathematica*. At the end of this string, append a `~&` to make the telnet window wait for a `&` after sending the command to start the kernel. When you have read what’s in the window, you can type a `&` to make it disappear. A couple of error messages are common in this circumstance. One is “math: Command not found.” Here, the command `math` was not found on your UNIX path, indicating that either *Mathematica* was not installed correctly on the UNIX machine, or the `math` file was not placed in a directory that is on the user’s path. The other common error message is from the kernel itself:

```
LinkConnect::linkc: LinkObject[3041@mosquito.wri.com, 1, 1]
is dead; attempt to connect failed.
```

It probably means that the linkname sent by the front end was not sufficient for the kernel to locate and connect back to it. If you look at the login string in the “Communications Toolbox Login” box, you will see that it ends with

```
-LinkName "$LinkName"\r
```

The front end will replace the variable `$LinkName` with the link’s port number, which is not known until after the front end opens its listening link. `$LinkName` is supposed to be replaced with something like `3041@kjmaw.wri.com`, or possibly `3041@140.177.10.186`. Sometimes, however, the code that does the replacement misbehaves and `$LinkName` resolves to just `3041` with no `@` and no IP address or machine name. You can look in the paused telnet window to see if that is what happened. If so, change the “`$LinkName`” portion of the login string to something like “`$LinkName@140.177.10.186`” to include the IP address of your Macintosh and try the connection again. This fix may not work in one circumstance. If you are running security software on your UNIX machine (something like *wrapper*), you must have the IP address of your Macintosh reside in the `/etc/hosts` file of your UNIX machine.

Symbolic Complex Algebra using ComplexExpand

Robby Villegas
villegas@wri.com

Suppose you have an expression like `Cos[x + y I]` and you want to know its real and imaginary parts, or its modulus and argument. The function `ComplexExpand` is designed to expand an expression involving functions with symbolic complex arguments:

```
In[1]:= ComplexExpand[ Cos[x + y I] ]
Out[1]= Cos[x] Cosh[y] - I Sin[x] Sinh[y]
```

Unlike other *Mathematica* functions, `ComplexExpand` assumes that all your variables are real and that all complex quantities are written explicitly in the form `x + y I`. This assumption makes it easy to identify the roles of the real and imaginary parts (or the moduli and arguments, or other components) of your original arguments in the expanded expression.

To exhibit the flavor of `ComplexExpand`, we look at a few classes of examples. In simple arithmetic, `ComplexExpand` will expand products and rationalize denominators, grouping real and imaginary parts in the result:

```
In[2]:= ComplexExpand[(x + y I)(a + b I)]
Out[2]= a x - b y + I (b x + a y)

In[3]:= ComplexExpand[ (a + b/I) / (c - d/I) ]
Out[3]=  $\frac{a c}{c^2 + d^2} - \frac{b d}{c^2 + d^2} + I \left( -\frac{b c}{c^2 + d^2} - \frac{a d}{c^2 + d^2} \right)$ 
```

`ComplexExpand` applies Euler’s formula to expressions of the form `Exp[x + y I]`:

```
In[4]:= ComplexExpand[ Exp[x + y I] ]
Out[4]= E^x Cos[y] + I E^x Sin[y]
```

`ComplexExpand` will compute a formula for a function applied to a complicated complex argument:

```
In[5]:= ComplexExpand[ ArcCos[(u + v I)/(a - b I)] ]
Out[5]=  $\frac{\text{Pi}}{2} - \text{Arg}\left[\text{Sqrt}\left[1 - \frac{(u + I v)^2}{(a - I b)^2}\right] + \frac{I (u + I v)}{a - I b}\right] + I \text{Log}\left[\text{Abs}\left[\text{Sqrt}\left[1 - \frac{(u + I v)^2}{(a - I b)^2}\right] + \frac{I (u + I v)}{a - I b}\right]\right]$ 
```

`ComplexExpand` also handles exact complex numbers:

```
In[6]:= ComplexExpand[ (1 - I)^(1/3) / (1 + I)^(2/3) ] // Simplify
Out[6]=  $\left(\frac{1}{2} - \frac{I}{2}\right)^{1/3}$ 
```

`ComplexExpand` attempts to produce a form using the simplest possible functions (such as `Log`, powers, and radicals) or a form in which the elementary functions don't have imaginary quantities in their arguments. To this end, it will recognize functions that can produce complex values even on real input. For instance, `Log` is complex-valued for negative numbers, and `ArcSin` is complex outside of the interval `[-1, 1]`:

```
In[7]:= ComplexExpand[ Log[r] ]
```

$$\text{Out[7]} = I \text{Arg}[r] + \frac{\text{Log}[r^2]}{2}$$

```
In[8]:= ComplexExpand[ ArcSin[r] ]
```

$$\text{Out[8]} = \text{Arg}[I r + \text{Sqrt}[1 - r^2]] - I \text{Log}[\text{Abs}[I r + \text{Sqrt}[1 - r^2]]]$$

Although `ComplexExpand` makes the convenient assumption that variables in the given expression are real, it lets you specify that a variable represents a complex quantity. If `z` is specified as complex, then components of it, such as `Re[z]` and `Abs[z]`, will appear in the result. For instance, to expand `Sin[z]` with `z` taken as complex:

```
In[9]:= ComplexExpand[Sin[z], {z}]
```

$$\text{Out[9]} = \text{Cosh}[\text{Im}[z]] \text{Sin}[\text{Re}[z]] + I \text{Cos}[\text{Re}[z]] \text{Sinh}[\text{Im}[z]]$$

`ComplexExpand` has one option, `TargetFunctions`, to specify standard operators like `Re` and `Abs` in terms of which the result should be expressed. There are six such operators: `Re`, `Im`, `Abs`, `Arg`, `Conjugate`, and `Sign`. Given any list of these operators, `ComplexExpand` will use only those to express the result; none of the others will appear in the output. The most commonly used pairs of operators are probably `{Re, Im}` and `{Abs, Arg}` since they lead to expressions involving Cartesian and polar coordinates. However, you can use any combinations you like. Compare the following expansions of `Log[x - y I]` using different target functions:

```
In[10]:= ComplexExpand[Log[x - y I]]
```

$$\text{Out[10]} = I \text{Arg}[x - I y] + \frac{\text{Log}[x^2 + y^2]}{2}$$

```
In[11]:= ComplexExpand[Log[x - y I], TargetFunctions -> {Re, Im}]
```

$$\text{Out[11]} = I \text{ArcTan}[x, -y] + \frac{\text{Log}[x^2 + y^2]}{2}$$

```
In[12]:= ComplexExpand[Log[x - y I], TargetFunctions -> {Sign}]
```

$$\text{Out[12]} = \frac{\text{Log}[x^2 + y^2]}{2} + \text{Log}[\text{Sign}[x - I y]]$$

```
In[13]:= ComplexExpand[Log[x - y I], TargetFunctions -> {Conjugate}]
```

$$\text{Out[13]} = \text{Log}\left[\frac{x - I y}{\text{Sqrt}[(x - I y)(x + I y)]}\right] + \frac{\text{Log}[x^2 + y^2]}{2}$$

You can see the relationships between operators by using `ComplexExpand` on one operator and specifying another as the target:

```
In[14]:= ComplexExpand[Sign[x + y I], TargetFunctions -> Arg]
```

$$\text{Out[14]} = \frac{x}{\text{Sqrt}[x^2 + y^2]} + \frac{I y}{\text{Sqrt}[x^2 + y^2]}$$

```
In[15]:= ComplexExpand[Arg[x + y I], TargetFunctions -> Sign]
```

$$\text{Out[15]} = -I \text{Log}[\text{Sign}[x + I y]]$$

Finding components like `Re`, `Im`, or `Abs` of an expression that won't otherwise resolve is easy with `ComplexExpand`. For example, you probably know that *Mathematica* will not simplify the expression:

```
In[16]:= Re[x + I y]
```

$$\text{Out[16]} = \text{Re}[x + I y]$$

because `x` and `y` may be complex. This behavior is frustrating for many users, but `ComplexExpand` will do the trick since it assumes the variable are real:

```
In[17]:= ComplexExpand[Re[x + I y]]
```

$$\text{Out[17]} = x$$

We can write a small function that will give us the real and imaginary parts of an expression as an ordered pair:

```
In[18]:= ReImParts[expr_] :=
```

```
ComplexExpand[#, TargetFunctions -> {Re, Im}]& /@
```

```
Through[{Re, Im} @ expr]
```

For example:

```
In[19]:= ReImParts[ Log[u - v / I] ]
```

$$\text{Out[19]} = \left\{ \frac{\text{Log}[\text{Cosh}[y]^2 \text{Sin}[x]^2 + \text{Cos}[x]^2 \text{Sinh}[y]^2]}{2}, \text{ArcTan}[\text{Cosh}[y] \text{Sin}[x], \text{Cos}[x] \text{Sinh}[y]] \right\}$$

As an application, we'll verify that the real and imaginary parts of `Sin` satisfy the partial differential equations known as the Cauchy-Riemann equations, a necessary condition for differentiability of a complex function.

```
In[20]:= {u, v} = ReImParts[ Sin[x + y I] ]
```

$$\text{Out[20]} = \{\text{Cosh}[y] \text{Sin}[x], \text{Cos}[x] \text{Sinh}[y]\}$$

```
In[21]:= {{ux, uy}, {vx, vy}} = Outer[D, {u, v}, {x, y}]
```

$$\text{Out[21]} = \{\{\text{Cos}[x] \text{Cosh}[y], \text{Sin}[x] \text{Sinh}[y]\}, \{-\text{Sin}[x] \text{Sinh}[y], \text{Cos}[x] \text{Cosh}[y]\}\}$$

```
In[22]:= CRmatrix = {{0, 1}, {-1, 0}};
```

```
In[23]:= {ux, uy} == CRmatrix . {vx, vy}
```

$$\text{Out[23]} = \text{True}$$

We can encapsulate these steps into a function to test the Cauchy-Riemann equations. The function takes as argument either a pure function (no variables) or an expression in two specified variables.

```

In[24]:= CauchyRiemannTest[func_] :=
Module[{x, y, u, v, partials},
  {u, v} = ReImParts[ func[x + y I] ];
  partials = Outer[D, {u, v}, {x, y}];
  Apply[Simplify[#1] == Simplify[CRmatrix . #2] &,
    partials] ]
In[25]:= CauchyRiemannTest[formula_, {x_, y_}] :=
Module[{u, v, partials},
  {u, v} = ReImParts[formula];
  partials = Outer[D, {u, v}, {x, y}];
  Apply[Simplify[#1] == Simplify[CRmatrix . #2] &,
    partials] ]

```

We test the functions Sin, Exp, and Tan[x + y I]:

```

In[26]:= CauchyRiemannTest[Sin]
Out[26]= True

In[27]:= CauchyRiemannTest[Exp]
Out[27]= True

In[28]:= CauchyRiemannTest[Tan[x + y I], {x, y}]
Out[28]= True

```

Memory Allocation under Windows

John Fultz
jfulzt@wri.com

Setting the memory allocation for *Mathematica* to run under low memory conditions in Windows is not hard if you understand how the Windows version of *Mathematica* uses memory. This note discusses memory use in Version 2.2.1 (and earlier) of *Mathematica*, and supplements the discussion of memory management in the *Mathematica User's Guide for Microsoft Windows* (pages *xvi–xix*). All future versions will have more flexible memory-handling methods because they will use a *MathLink* connection between the front end and the kernel. In particular, the current 16MB limit on the kernel will disappear.

On a Windows machine, as on a NeXT or Macintosh, *Mathematica* is split into two programs, the front end and the kernel. The front end handles the interface chores, such as saving and loading notebooks, setting fonts and styles, printing, and rendering PostScript graphics. The kernel does all the computations. Anything that is evaluated is sent to the kernel to be processed.

This separation into two programs explains some of *Mathematica*'s behaviors that may at first seem odd. For example, by default the kernel is not loaded when you double-click the *Mathematica* icon. The reason your first calculation always takes a long time is that the kernel program must first be loaded into memory.

The *Mathematica* kernel allocates a block of 32-bit addressable memory to load itself in and store its data. There are two restrictions on how this allocation is made. The request for a contiguous block of memory can only be made when the kernel is loaded (so the program cannot ask for "a little more memory" later) and the request cannot exceed

16MB. Therefore, a decision must be made ahead of time of how much memory the kernel will need.

The amount of memory available depends, of course, on the machine. Some machines may not have enough available memory to give the *Mathematica* kernel even 8MB, while others may be able to give the full 16MB to the kernel and have plenty left over for the front end and other programs. Since the availability of memory varies widely, *Mathematica* lets users set the requested allocation for the kernel. The item called **Kernel...** under the **Options** menu pulls up a dialog box in which you can set the memory allocated to the kernel data. This memory includes both virtual and physical memory. You can check the total amount of memory available on your machine by selecting **About Program Manager** under the **Help** menu of the Program Manager. Remember that the memory you specify in the dialog box is only for kernel data. The kernel executable takes up about 4MB more. Therefore, the maximum amount of memory you can allocate is about 12MB.

When selecting the memory in the **Kernel Settings** dialog box, it's important to remember a few things. First, large calculations, such as those involving large tables, Fourier transforms and other complex mathematical operations, and some external packages, eat up memory quickly. If you plan to do large calculations, you should make sure the kernel has at least 8MB of memory (or even the full 12MB) for data. If you don't expect to do much more than integration, you can get by typically with about 6MB allocated to the kernel. If your calculations involve only algebra, 4MB may be sufficient.

When the kernel is loaded, all memory for the kernel executable and the data memory allocated to it from the **Kernel Settings** dialog box is removed immediately from the "memory free" specification at the bottom of the *Mathematica* window. The memory remaining is all that is available to the front end and other Windows programs. Therefore, you also must consider the front end's memory requirement when setting the kernel allocation. Remember that complex graphics take up lots of space in the front end. For example, a complex and intricate three-dimensional plot in your notebook can use over 500K of memory. If you plan to have lots of graphics in your notebooks, you should make sure the front end has the memory to store them. One possible way to accomplish this is to decrease the kernel memory allocation, which will increase the memory available to the front end.

A good rule of thumb is to give as much memory to the kernel as you can, leaving at least 2MB for the front end. That means the meter at the bottom of the *Mathematica* window reads about 2MB when no notebooks are open. However, when working under extreme low memory conditions or when doing complicated calculations, this may not be possible. In such cases, it's important to judge how much memory the kernel needs for the operations you will be performing.

Finally, remember that settings in the **Kernel Settings** dialog box do not take effect until the next time the kernel is started. The best way to find the right memory settings for your work may be to try out different settings, restart *Mathematica*, and watch what happens. It shouldn't take long to figure out the setting that will best fit your kind of work. 