Lab01

September 7, 2018

1 Lab 01 - CoCalc introduction

1.0.1 *Replace this text with your names!*

Cocalc (CoCalc.com) combines the capabilities of a word processor and a sophisticated calculator. This is a quick introduction to get you started. I assume you have already gone through the first two mathphys CoCalc links which show you how to sign up for an account and get started with a notebook.

1.0.2 Cells

There's a dropdown menu above that lets you choose between `Markdown` and `Code`: + This cell is a `Markdown` cell, used for **writing**. + The cell below is a `Code` cell used for **calculations**, or more generally for coding. See the `sin(...)` example below, + There's also a `Raw` option on that menu, which we'll ignore for the time being.

The language (or 'kernel') that you're using for code cells is also shown in the menu bar above. Yours probably says "**SageMath 8.3**" right now, e.g.

`&nbsp;`

1.1 Calculations with `Code` cells

The code cell below calculates $\sin^2\left(\frac{2\pi}{3}\right)$ using the current kernel--**SageMath**. Observe that + SageMath has names for many constants (`pi` for π). + Most of these are *case sensitive*. So you won't get the same effect from `Pi` as `pi`. + You must use `*` for multiplication. The caret, `^`, for exponentiation. + The square appears at the end because $\sin^2(2\pi/3)$ is shorthand for $(\sin(2\pi/3))^2$.

To carry out the calculation below, + Click your mouse **anywhere in the cell** (not just at the end of the line) and + then press the `Shift+Enter` keys together.

A new "output" cell will be created with the answer.

```
In [1]: (
            sin(  2*pi/3  )
        )^2
```

```
Out[1]: 3/4
```

[By the way, work through the rest of this notebook, hitting `Shift+Enter` in each input cell in turn to see the result.]

- Notice in the cell above that you can add extra spaces and line breaks (use these for readability) and they'll be ignored for purposes of calculation.
- You get an **exact** answer (the fraction 3/4) unless you indicate otherwise. You can request an *approximate* answer by placing your expression inside the brackets of the `N(...)` function, or putting a decimal point in to any of the numbers. E.g., 3.0 indicates an approximate number whereas 3 indicates an exact number.

- Click your mouse anywhere in the cell below and press the `Shift+Enter` keys together. A new output cell will be created with the answer to *only the last expression* of the 3 entered below:

```
In [2]: # This is a comment in a code cell.
        #
        # There are 3 expressions here (each on their own line)
        # but only the last one should result in a printed, numerical value
        #
        (sin(2*pi/3))^2
        N(   (sin(2*pi/3))^2   )
        N((sin(2.*pi/3))^2)
```

```
Out[2]: 0.750000000000000
```

1.1.1 By default Sage only prints the output of the last command, if you have multiple commands.

To print several results, enclose all your commands (separated with commas) within **sqare brackets**. This structure is called a *list* and will print out a list of answers, one for each expression. Here are 3 results:

```
In [3]: [  (sin(2*pi/3))^2,  N(   (sin(2*pi/3))^2   ),  N( (sin(2.*pi/3.))^2 )   ]
```

```
Out[3]: [3/4, 0.750000000000000, 0.750000000000000]
```

Exercise 1: Explain in words what is being calculated in each of the 4 expressions below. You should explicitly recognize what *base* is being used in each logarithmic expression. (For example, the first one is calculating the logarithm (base e) of the exponential constant e. That is to say, the answer is 1 because e is equal to e^1):

1. Use the plus button on the bottom toolbar, , to insert a new cell.
2. Make it a `Markdown` cell to write out your explanations. You will use the `Shift+Enter` key combination to "render" your markdown cell.
3. Move your markdown cell after the calculations cell using the up and down arrow keys as necessary on the toolbar.

```
In [5]: [  ln(e),   log(e), log(100.),   log(100.,10)  ]
```

```
Out[5]: [1, 1, 4.60517018598809, 2.00000000000000]
```

In general, for each of the following exercises you'll need to insert one or more markdown and/or code cells to carry out a calculation and answer any questions posed.

Exercise 2: Insert one or more `Code` cell(s) below and use them to calculate an exact expression for, and an approximate, numerical value for $\sqrt{e^3 + \tan(\pi/6)}$, where e is the exponential constant. Hint: In the exact expression, e^3 cannot be simplified any further. But there is an exact numerical simplification for $\tan(\pi/6)$.

1.2 Names in SageMath

Any string of letters and numbers can represent a **named object** in SageMath. In the following cell the *value* 7.4 is *assigned* to the *named object* `SomeNumber`. The second line, just the name, causes the contents of `SomeNumber` to be printed out:

```
In [6]: SomeNumber=7.4
        SomeNumber
```

```
Out[6]: 7.40000000000000
```

You can calculate with `SomeNumber`:

```
In [7]: SomeNumber-2.4
```

```
Out[7]: 5.00000000000000
```

Expressions You can also assign an "expression"--an abstract combination of symbols--to a named object. For example:

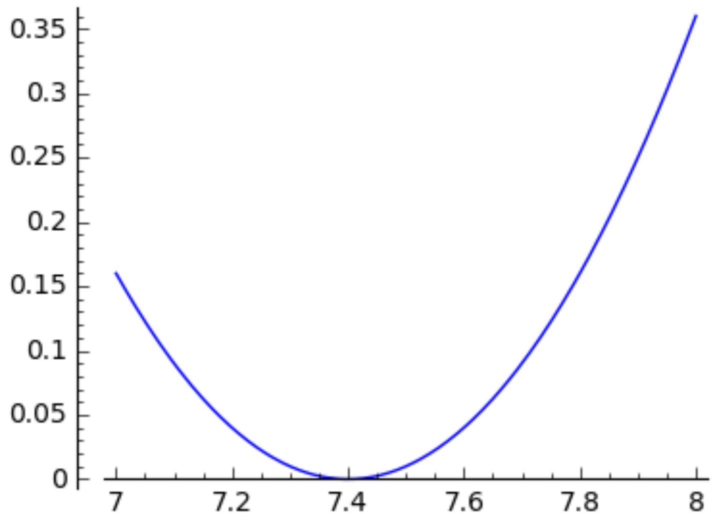
```
In [8]: MyExpression = (x-SomeNumber)^2
        MyExpression
```

```
Out[8]: (x - 7.40000000000000)^2
```

And you can plot an expression...

```
In [12]: plot(MyExpression, 7, 8, figsize=4)
```

```
Out[12]:
```



Variables The named object x is different from `SomeNumber`. Instead of just having one value, it is a placeholder, that takes on different value in the `plot(...)` expression. It is a **variable**.

In SageMath, x is special in that it is already set up as a variable. That is to say, execute this next line which uses t instead of x and *you will get an error*:

```
In [13]: MyExpression = (t-SomeNumber)^2
```

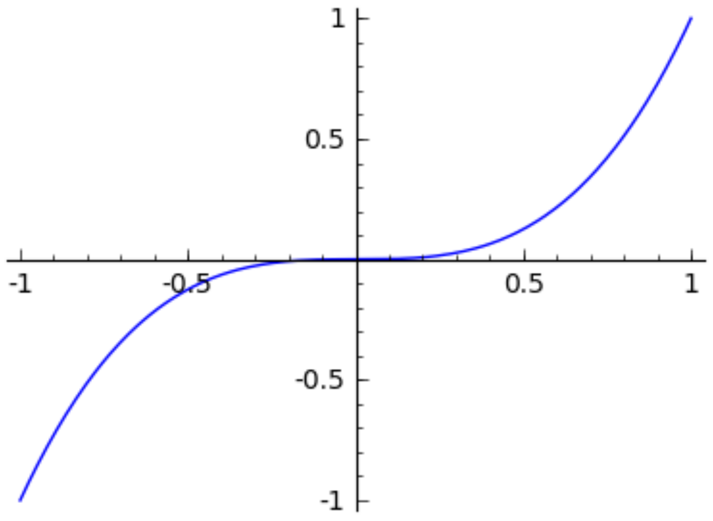
```
    ---------------------------------------------------------------------------

    NameError                                 Traceback (most recent call last)

    <ipython-input-13-ef98e1eb446e> in <module>()
----> 1 MyExpression = (t-SomeNumber)**Integer(2)


    NameError: name 't' is not defined
```

Anything other than x that you want to be treated as a variable has to be **declared as a variable** before you can use it in expressions / plot it / etc.

Here's an example of declaring a couple of variables at one time, and then using one of them to make a plot:

```
In [14]: myx,y,t=var('myx,y,t')
         plot((myx)^3, -1,1, figsize=4)
```

```
Out[14]:
```



And now this assignment should work without a hitch:

```
In [15]: MyExpression = (t-SomeNumber)^2
         MyExpression
```

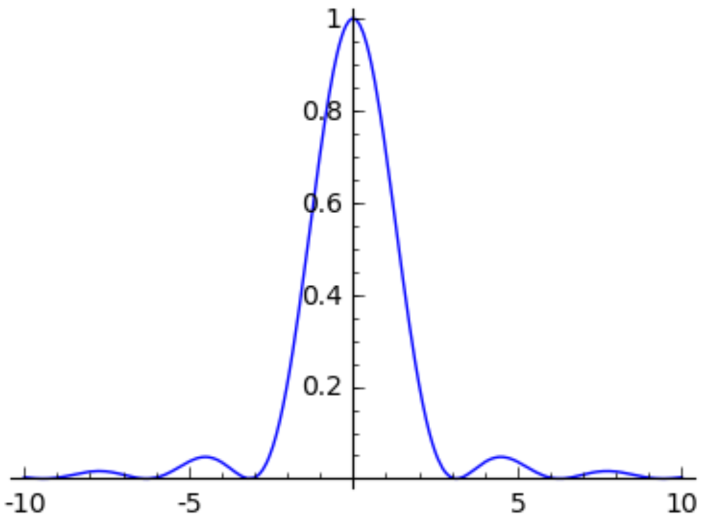
```
Out[15]: (t - 7.40000000000000)^2
```

Exercise 3: Declare u as a variable, then assign this expression a name and plot it for u values from -10 to +10.

$$\frac{\sin^2(u)}{u^2}$$

```
In [16]: u=var('u')
         myf = (sin(u)/u)^2
         plot(myf,-10,10, figsize=4)
```

```
Out[16]:
```



Functions You are probably more used to plotting functions. This is how to declare a function: (You don't have to declare `f` ahead of time...)

```
In [17]: f(x)= sin(x)/x
         f(x)
```

```
Out[17]: sin(x)/x
```

You can evaluate the function for a particular x value. Notice the different output with and without the decimal point:

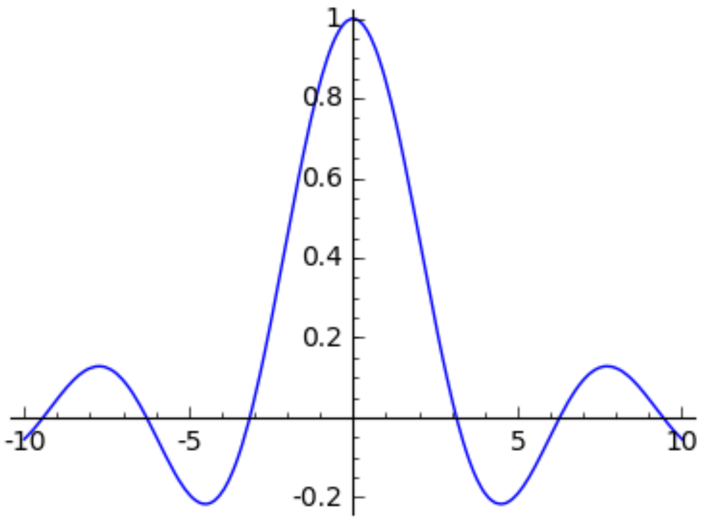
```
In [18]: [    f(3),     f(3.0)    ]
```

```
Out[18]: [1/3*sin(3), 0.0470400026866224]
```

Plot the function over a particular domain...

```
In [19]: plot(f(x),-10,10, figsize=4)
```

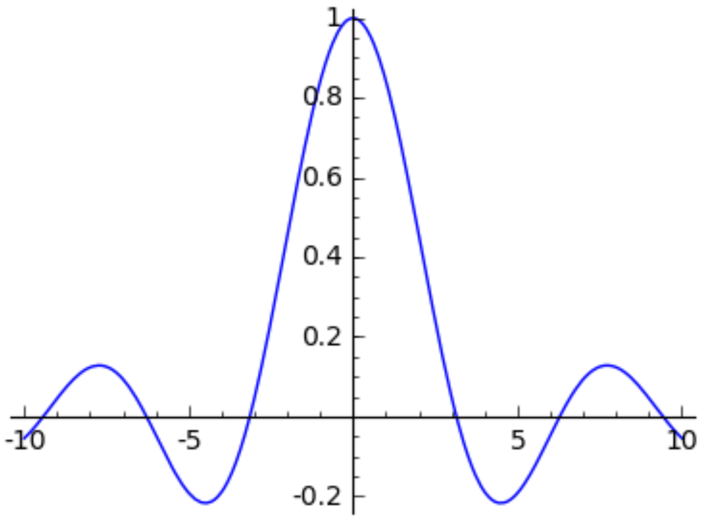
```
Out[19]:
```



Change the plotting variable (to a different one *which you have previously declared to be a variable*):

```
In [20]: plot(f(u),-10,10, figsize=4)
```

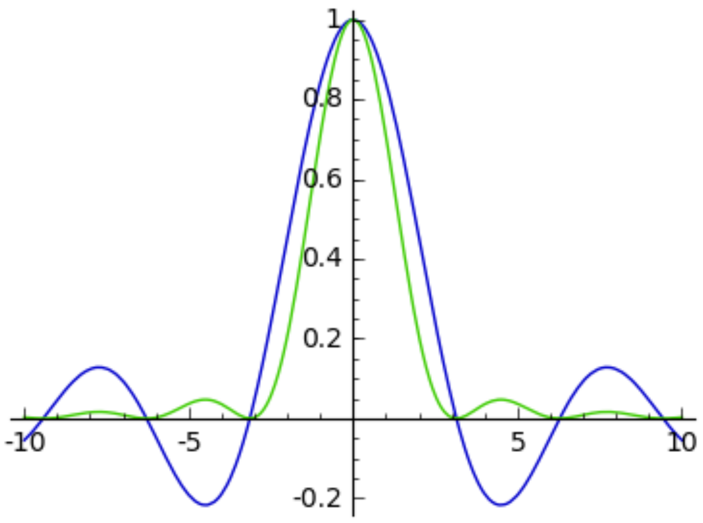
```
Out[20]:
```



And plot a *list* of functions to have more than one function plotted on the same graph:

```
In [21]: plot(    [f(u),myf],
             -10,10, figsize=4
             )
```

```
Out[21]:
```



Function composition Now, you can also do function composition. For example, defining

$$g(x) = x^2$$

We can get expressions for compositions...

```
In [22]: g(x)=x^2
         [  f(x), g(x),    f(g(x)),    g(f(x))]
```

```
Out[22]: [sin(x)/x, x^2, sin(x^2)/x^2, sin(x)^2/x^2]
```

Exercise 3: Define a function $h(x)$ to be

$$h(x) = x + 3.$$

Calculate by hand what you expect for $h(g(x))$ and $g(h(x))$. Then calculate with SageMath what you'd get. Do they agree?

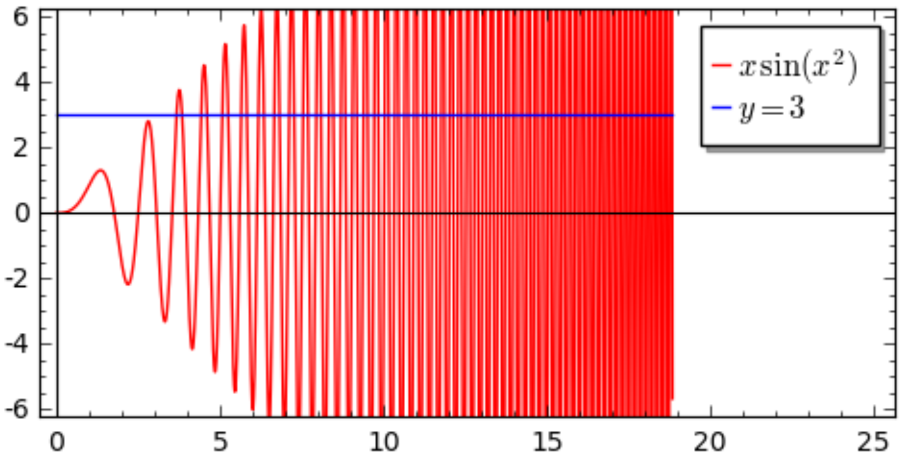
```
In [25]: h(x)=x+3
         [h(x),h(g(x)),g(h(x))]
```

```
Out[25]: [x + 3, x^2 + 3, (x + 3)^2]
```

***Lots* of plotting options** You can get an example of `plot(...)` with *many* options from the "Assistant" (see the top right of the screen). Here's an example showing several of the things you can do with plots, including 1. Using `show( pt+just3,...)` to combine named plots. 1. Using LaTeX to print using math typesetting.

```
In [26]: pt     = plot( x*sin(x^2), x,  0, 6*pi, rgbcolor="red",  linestyle = "-",  fill=False,

         just3 = plot( 3, x, 0, 6*pi, rgbcolor="blue", legend_label="$y=3$")

         show( pt+just3,  aspect_ratio = 1, axes=true, frame=True,  gridlines=false, figsize=5
```

```
Out[26]:
```



Equations...and solving them

In SageMath the single '=' is actually the *assignment operator*. So x=7 assigns 7 to the variable x. But assigning x to 7 makes no sense so 7=x will throw an error.

The = in an equation such as

$$x = 7 - x$$

expresses a relationship, **equality**, between two *expressions*. So, this sense of '*equals*' gets a different symbol in SageMath, namely **==**. You can *solve* the equation above for x like this:

```
In [27]: solve(   x==7-x,    x    )
```

```
Out[27]: [x == (7/2)]
```

You can assign equations to named objects, and solve systems of equations (using lists...):

```
In [28]: eq1=  x+y==10
         eq2=  y-x==2
         solve(  [eq1, eq2], [x,y])
```

```
Out[28]: [[x == 4, y == 6]]
```

Exercise 4: Use `solve(...)` to solve the quadratic equation

$$x^2 - 5x = -6$$

for x. [Hint: '5x' will mean nothing to SageMath. Code that piece of the expression differently to express multiplication.] Does the answer (or answers) you get make sense?

```
In [29]: solve(x^2-(5*x)==6,x)
```

```
Out[29]: [x == 6, x == -1]
```

1.3 Data

Let's say that you're a forester and have some data on a set of 8 ponderosa pines:

1. the waist high **circumference** (in inches) of each of the 8 pines.
2. and the **usable lumber** (in board feet) of each.

You'll start by making one *list* for the circumference data, and another one for the usable lumber data.

```
In [30]: cdata=[5.4,6.4,7.3,8.9,10.2,11.5,12.1,13.1]
         udata=[190,320,570,1130,1230,1920,2520,2940]

         table( rows=[cdata,udata], header_column=['circumference (inches)', 'usable lumber (bo
```

Now, make a list of data points... the `zip(...)` command pairs each value from `cdata` with the corresponding value in `udata`.

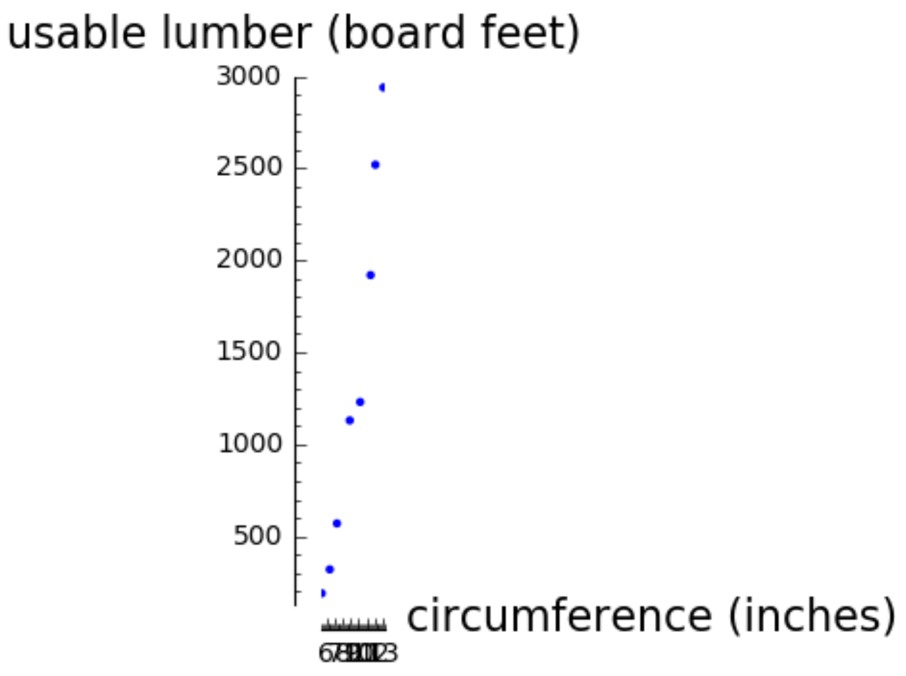
```
In [31]: treedata=zip(cdata,udata)
         treedata
```

```
Out[31]: [(5.40000000000000, 190),
          (6.40000000000000, 320),
          (7.30000000000000, 570),
          (8.90000000000000, 1130),
          (10.2000000000000, 1230),
          (11.5000000000000, 1920),
          (12.1000000000000, 2520),
          (13.1000000000000, 2940)]
```

Plot those points, and label the resulting graph axes.

```
In [35]: list_plot(treedata,axes_labels=["circumference (inches)","usable lumber (board feet)"]
```

```
Out[35]:
```



Fitting a model to data Let's fit the data to a linear model, that is...

$$\text{linearfit}(x) = m * x + b$$

We'd like to find the best values for the slope, m, and y-intercept, b.

```
In [37]: # design a model with adjustable parameters m and b.
         m,b=var('m, b')
         linearmodel(x) = m*x + b

         # find the parameters that best fits the model
         solution = find_fit(treedata, linearmodel)
         show(solution)
```

`solution` is a *list* with two elements.

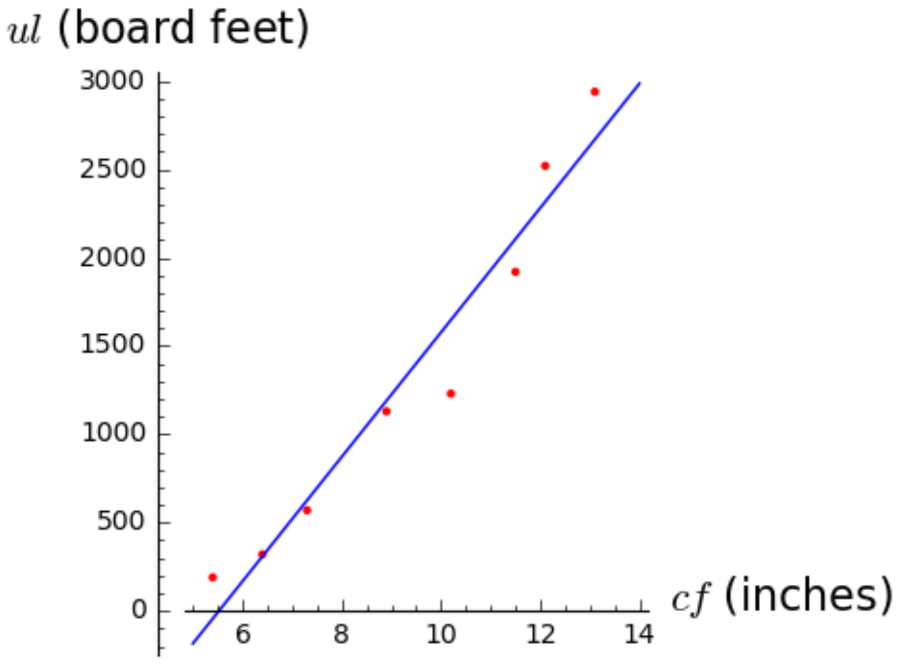
- Element 0 (the first element) is "b=(-1946.9....)".
- The number that we'd like to use in our model is on the *right hand side* of the equals sign in that first element.

See how we use this in what follows to avoid re-typing the numbers for our fitted model...

```
In [41]: # define a function, linfit(cf), the model with the parameters set to the fitted valu
         linfit(x) = linearmodel(b=solution[0].rhs(),m=solution[1].rhs())

         # create an empty plot object
```

```
linplot = plot([])
# add a plot of the model, with respect to cf from 5 to 14
linplot += plot(linfit(x),x,[5,14])

# add a plot of the data, in red
linplot += list_plot(treedata,color='red')
show(linplot,
    axes_labels=["$cf$ (inches)","$ul$ (board feet)"], figsize=5
    )
```

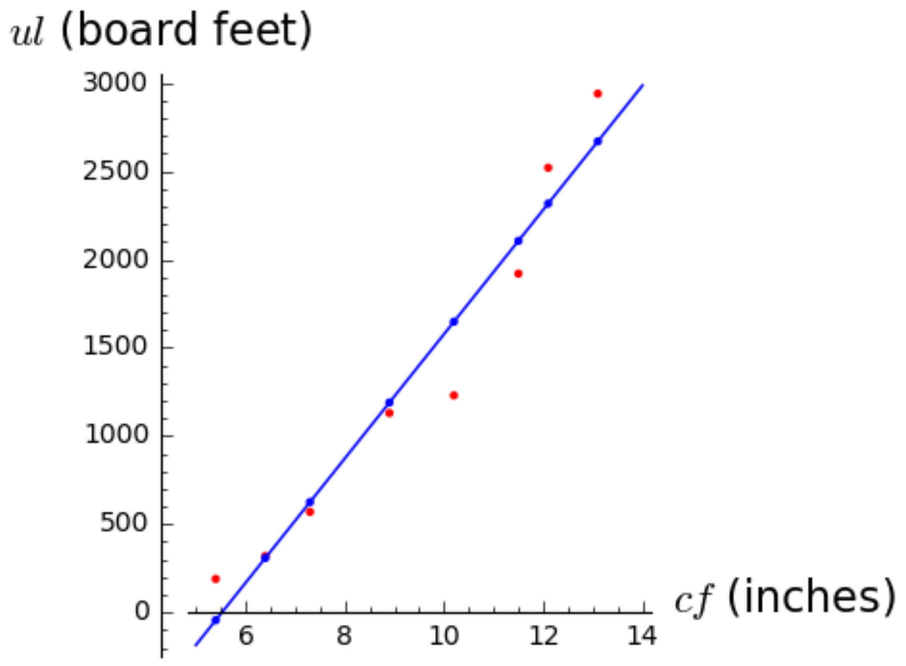
Out[41]:



How good is the fit? Error calculations... In order to compare this model to another model, we should calculate how far away the data points are from the fit line. And then do some statistics on those distances. In the graph below, we'll plot the predicted value according to the model (in blue) and the observed values (in red).

In [42]:
```
# Make a list of the model value for y corresponding to each x value
upredicted=[linfit(i) for i in cdata]

# And then add another data set to the graph we already produced, showing the predict
linplot += list_plot(zip(cdata,upredicted))
show(linplot,
    axes_labels=["$cf$ (inches)","$ul$ (board feet)"], figsize=5
    )
```

Out[42]:



The 'error' for each data point is the distance from the u data point to the u value as predicted from the fit.

```
In [0]: errors=list(vector(upredicted)-vector(udata))
        errors=[abs(i) for i in errors]
        errors
```

Look at the list of errors and compare to the graph. Does each error look about right?

Below are three kinds of "goodness of fit" statistics. See if those seem to make sense by comparing to the list of errors.

```
In [0]: [
        # Here's the mean (or average) error:
            mean(errors),

        #Here's the root mean square error (square each error, average, then take the square r
            sqrt(mean([i^2 for i in errors])),

        # And here is the maximum error value
            max(errors)

            ]
```

Exercise 5: Go through the same procedures above to fit the data to a *quadratic* function instead. That is:

$$y(x) = a * x^2 + b * x + c.$$

Graph the data along with the fit, and find the "goodness of fit" statistics for this model. Comment on which model (linear or quadratic) appears to fit the data better.

Exercise 6: Use whichever model you think works best to **predict** how many board feet you think you would get from a tree with a circumference at waist height of 15 inches!

Exercise 7: Reflect a bit on what you have done. What is some practical advice that you could give to your "earlier self" to make it easier to do what you have now done?