



# Programming inside TinkerCell

<b>1. Introduction .....</b>	<b>2</b>
<b>2. Submitting to the Repository .....</b>	<b>2</b>
<b>3. How TinkerCell loads C programs .....</b>	<b>4</b>
<b>4. Available C functions .....</b>	<b>4</b>
<b>5. Integrating C programs in TinkerCell .....</b>	<b>6</b>
• Option 1: drag and drop .....	6
• Option 2: button and menu item.....	7
• Option 3: callback functions .....	8
• Option 4: interfacing via a C++ plug-in .....	8
<b>6. Available Python functions .....</b>	<b>8</b>
<b>7. Integrating Python scripts .....</b>	<b>9</b>
<b>8. Available Octave functions.....</b>	<b>10</b>
<b>9. Integrating Octave script.....</b>	<b>11</b>
<b>10. Available Ruby functions .....</b>	<b>12</b>
<b>11. Integrating Ruby script.....</b>	<b>12</b>
<b>12. Example Code .....</b>	<b>12</b>
• Perturb all RBS's in the model (using python).....	13
• Testing for non-monotonic input/output response (using Octave) .....	14
• Optimize for non-monotonic input/output response (using python).....	15

- Perturb all parameters with a specific prefix (using python). 17
- Set global parameters (using python or octave) ..... 19

## Introduction

The TinkerCell application is a graphical user interface that serves as a "host" to third-party C programs. The ability to integrate C programs automatically implies integration of other C-based languages such as Python, Perl, Ruby, and R. Currently, Python, Octave, and Ruby have been integrated, although only Python is fully tested in all major platforms. Other languages such as R have not been integrated yet, but it should be possible if required.

The TinkerCell API uses C data structures (described [here](#)). For this reason, the functions do not return Python or Octave data structures. However, functions named "toTC" and "fromTC" are made available for converting between C data structures and the languages' data structures (see example code below).

All TinkerCell functions begin with "tc\_", e.g. tc\_insert. For Octave and Ruby, the scripts would also need the module name in front, e.g. tinkercell.tc\_insert. In Python, this is not necessary if you use the import statement: from tinkercell import \*

## Submitting to the Repository

[tinkercellextra.sf.net](http://tinkercellextra.sf.net) is the Sourceforge repository for TinkerCell plug-ins and module files. To submit a plug-in or module (model), you need: (1) a Sourceforge account and (2) install [Subversion](#) on your computer. Then contact someone who has access to tinkercellextra.sf.net so that you can be added to this group. Once you are part of the group, use the following

command to download the existing plug-ins (for Windows, it might be easier using [TortoiseSVN](#). See this [video](#)):

```
svn checkout https://tinkercellextra.svn.sourceforge.net/svnroot/tinkercellextra  
TinkerCellPlugins
```

The TinkerCellPlugins directory will be created with all the existing plug-ins and module files. Go into the folder. If you want to add a new Python plug-in, go into the "python" folder and add the new plug-in file there. If you want to add octave plug-in, add the file into the "octave" folder. For C plug-ins, you must also select the right folder for your operating system. After adding the new file, use the following commands to add and upload the new file to the repository:

```
svn add MyNewPluginFile  
svn commit
```

Replace "MyNewPluginFile" with the name of your plug-in file. You must be inside the directory when executing these commands. Do the above two commands for each file you want to add to the repository. You can modify your code later and submit the modified code by doing:

```
svn commit
```

And you can use the following command to get all the updates from other plug-ins:

```
svn update
```

Note that all the above commands must be made inside the TinkerCellPlugins folder (or whatever folder name you are using). If you need to know more details, read about Subversion.

## How TinkerCell loads C programs

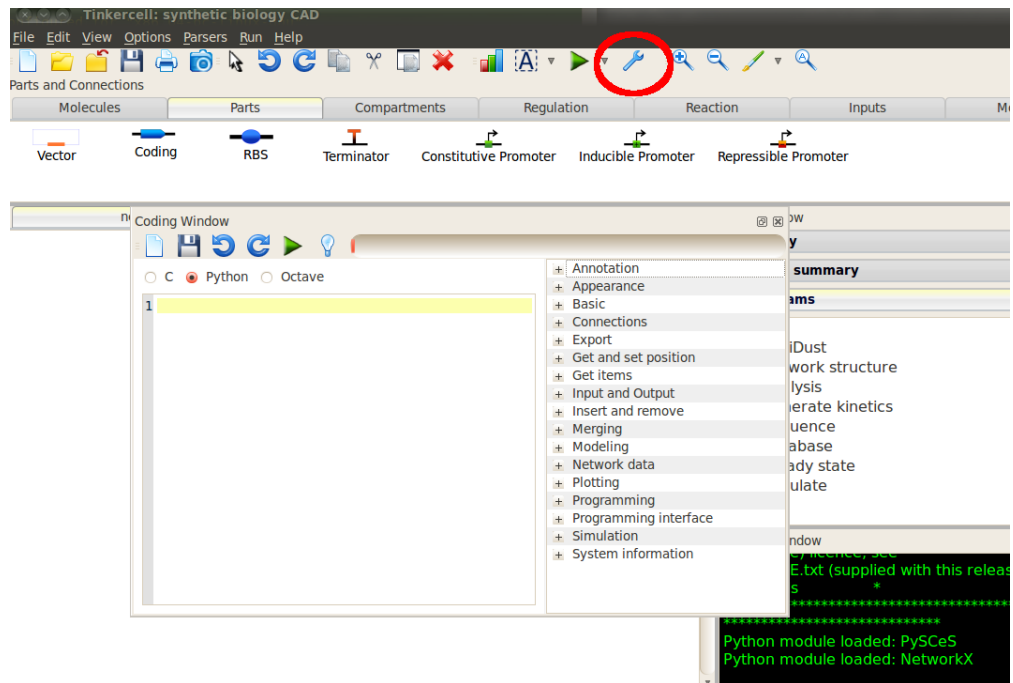
All C programs that are loaded into TinkerCell must be **dynamic libraries**. The dynamic libraries must be placed in the **Plugins/c** of the TinkerCell directory. TinkerCell will look for the **void tc\_main()** function in the library and call that function.

**void tc\_main()** is the main entry point. If it is not defined, nothing will happen when the library is loaded, **unless the library has an accompanying C++ plug-in** that is already loaded into TinkerCell. For example, the flux balance analysis plug-in in TinkerCell is a combination of a C++ plug-in, which provides the graphical user interface, and a C library, which provides the linear optimization function. The graphical interface knows which functions exist in the C library and calls those functions specifically; therefore tc\_main does not have to be defined in such cases.

## Available C functions

The C application programmer interface (API) is located at: <http://tinkercell.sourceforge.net/CAPI/index.html>

Another resource for studying is the API is to run TinkerCell and open the "Coding Window" (image shown to the right). The Coding Window



contains a list of all the functions available in the C API, organized by categories. Double-clicking on one of the functions will place an example call to the function on the text editor.

# Integrating C programs in TinkerCell

Suppose we are interested in integrating the following trivial function into TinkerCell:

```
double function(double a, double b)
{
    return (a + b);
}
```

There are four options for integrating the function into the user interface:

1. The user will drag and drop the file into TinkerCell. TinkerCell will run the function and then unload the library. It will not remain loaded in TinkerCell.
2. The function will appear as a button in TinkerCell. The library will remain loaded until TinkerCell exists.
3. The library uses callback functions to interact with TinkerCell, i.e. no buttons are available for the user to click. The library will remain loaded until TinkerCell exists.
4. The library has an accompanying C++ plug-in in TinkerCell. The C++ plug-in manages all the calls to the C library.

Most of these options can be used in combination as well. For example, a library can have callback functions as well as buttons for user interaction.

## Option 1: drag and drop

When a C dynamic library file is dropped on the TinkerCell main window, TinkerCell will look for "void tc\_main()" and call that function. So, in this case, the C code would be as follows:

```
#include "TC_API.h"

double function(double a, double b)
{
```

```

        return (a + b);
    }

TCAPIEXPORT void tc_main()
{
    double d, x=10.0, y = 5.0;
    char c[100];
    sprintf(c,"answer = %lf\0", function(x,y));
    tc_print(c);
}

```

Compile the above code as a shared (dynamic) library (wikipedia has a nice article on [shared libraries](#) if needed). Be sure to list the Tinkercell/API directory in your set of include directories when compiling. That is where TC\_api.h is located. TC\_api.h includes all the other header files from the API folder, so all the other header files are also necessary.

## Option 2: button and menu item

Use the `tc_addFunction(&f, "name of function", "short description", "category", "icon file", "restrictions", int, int, int);`

The "restrictions" argument can be used so that this function only shows up when items of that family are selected. For example, if "promoter" is used for the "restrictions" argument, then this function will only appear when promoter nodes are selected.

The last three arguments are either 1 or 0 for answering the following:  
 insert button in functions list? insert button in menu? make this the primary function?

See `runcvode.c` for example. Look at the `tc_main` function.

### **Option 3: callback functions**

Use `tc_callback( void (*f)(void) )` to assign a function as the callback function. The function `f` will get called whenever the model is altered, i.e. when items are inserted, deleted, renamed, or when data within items are changed.

...more documentation to come...

### **Option 4: interfacing via a C++ plug-in**

This is primarily done using a C++ class that inherits from `AbstractInputWindow` class, one of the `TinkerCellCore` classes. `AbstractInputWindow` can provides interface for loading a library and connecting signals to functions in that library. The C++ class would setup its widgets so that the buttons or other interfaces in the widgets trigger functions in the library. All calls will be done using a `CThread` (i.e. multi-threaded). See the `OtherTools/LPSolveInput.cpp` for a simple example of a C/C++ combined plug-in.

## **Available Python functions**

The set of Python functions is the same as the set of C functions, which is located at: <http://tinkercell.sourceforge.net/CAPI/index.html>

Using the "Coding Window" (same image shown previously) is another good option for learning all the available functions and testing a new script. The Coding Window contains a list of all the functions available in the Python API. Double-clicking on one of the functions will place an example call to the function on the text editor. The pressing the "Play" button on top of the editor will run the script.



Two important functions are `fromTC` and `toTC`. These functions are used to convert from and to TinkerCell data types. For example:

```
a = tc_allItems()  
m = tc_getStoichiometry(a)  
M = fromTC(m)    #get python 2D list
```

## Integrating Python scripts

Integrating a Python code requires two things:

1. adding TinkerCell header on top of the Python script
2. copying the .py file inside <Documents>/**TinkerCell/python** folder or <TinkerCell installed directory>/**python** folder

The TinkerCell header looks like this:

```
"""  
category: category for the new python script  
name: name of the python function  
description: description of python function  
icon: location of icon relative to <TinkerCell installed directory>  
menu: yes or no, indicating whether or not to place the function in the  
functions menu  
specific for: optional entry to indicate whether this function is specific for a  
type of objects  
tool: yes or no indicating whether or not a tool button should appear when  
the specific objects are selected  
"""
```

Here is an example from the `hillEquations.py` script, which is a function that specifically works on "Synthesis" type reactions (Synthesis is a parent family for transcription).

"""

category: Generate kinetics

name: Hill equations

description: automatically generate the equilibrium rate equation for  
transcript

ion

icon: plugins/c/hillequation.png

menu: yes

specific for: Synthesis

tool: yes

"""

## Available Octave functions

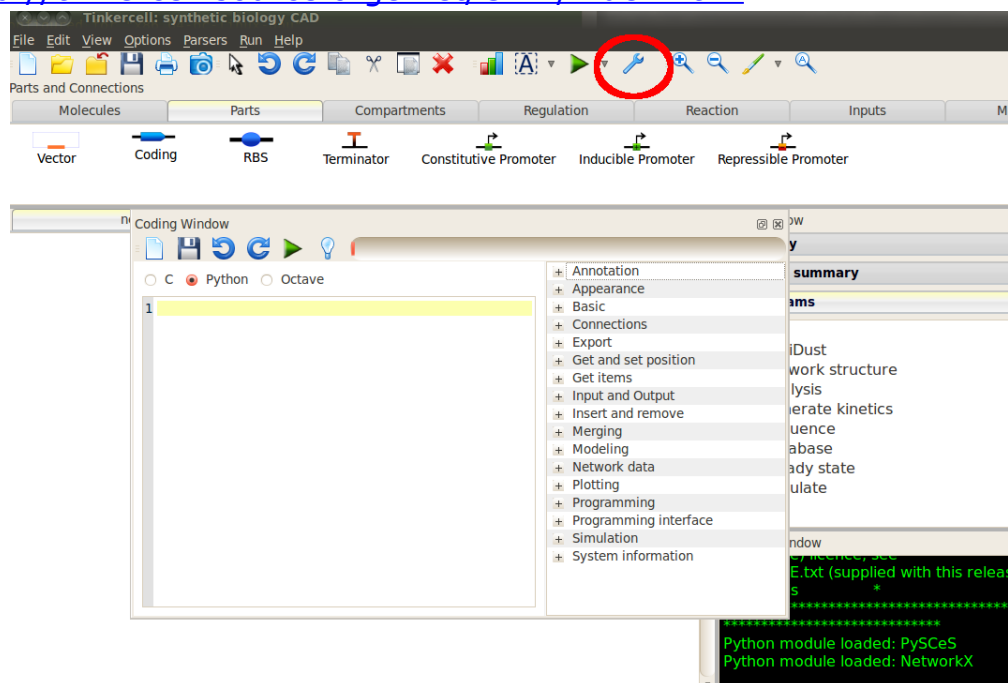
The set of Octave functions is the same as the set of C functions, which is located at: <http://tinkercell.sourceforge.net/CAPI/index.html>

Using the  
"Coding  
Window"

(image  
shown to the  
right) is  
another good  
option for  
learning all  
the available  
functions  
and testing a  
new script.

The Coding

Window contains a list of all the functions available in the Octave API.



Double-clicking on one of the functions will place an example call to the function on the text editor.

Pressing the "Play" button on top of the editor will run the script.

## Integrating Octave script

Integrating Octave code requires two things:

1. adding TinkerCell header on top of the Octave script
2. copying the **.m** file inside <Documents> **/TinkerCell/octave** folder or <TinkerCell installed directory> **/octave** folder

The TinkerCell header looks like this:

```
%category: category for the new python script
%name: name of the python function
%description: description of python function
%icon: location of icon relative to <TinkerCell installed directory>
%menu: yes or no, indicating whether or not to place the function in the
functions menu
%specific for: optional entry to indicate whether this function is specific for a
type of objects
%tool: yes or no indicating whether or not a tool button should appear when
the specific objects are selected
```

An important function is **fromTC** , which is used to convert from TinkerCell data types. **For MS Windows, fromTC will only work on tc\_matrix.** Additionally, due to complications in MS Windows, the Octave scripts **must use the tinkercell prefix**; for example:

```
a = tinkercell.tc_allItems()  
m = tinkercell.tc_getStoichiometry(a)  
M = fromTC(m) %get octave matrix  
rank(M)  
m = tinkercell.tc_getPlotData(0)  
M = fromTC(m)  
cov(M)
```

## Available Ruby functions

The set of Ruby functions is the same as the set of C functions, which is located at: <http://tinkercell.sourceforge.net/CAPI/index.html>

## Integrating Ruby script

Integrating Ruby code requires two things:

1. adding TinkerCell header on top of the Ruby script file
2. copying the **.rb** file inside <Documents>/**TinkerCell/ruby** folder or <TinkerCell installed directory>/**ruby** folder

Ruby integration works the same way as Octave and Python integration mentioned above

## Example Code

## Perturb all RBS's in the model (using *python*)

```
from tinkercell import *

#get all RBS parts
rbs_parts_c = tc_itemsOfFamily("RBS")
rbs_names_c = tc_getUniqueNames(rbs_parts_c)

#get just RBS related parameters
rbs_params = tc_getParameters(rbs_parts_c)

#rbs perturbation
for n in range(0, rbs_params.rows):
    tc_showProgress("mRNA Perturbation", int(100 * n /
rbs_params.rows))
    x = tc_getMatrixValue( rbs_params , n , 0)
    tc_setMatrixValue(rbs_params, n, 0, 0.2 * x)
    tc_updateParameters(rbs_params) #temporary update (faster
than setParameters)
    sim = tc_simulateDeterministic(0,500,500)
    s = ""
    for j in range(0, sim.cols):
        col = str(tc_getColumnName(sim, j))
        if (j == 0):
            s += col
        else:
            p = tc_find(col)
            if tc_isA(p, "mRNA") == 1: #only output mRNA data
                s += "\t" + str(tc_getColumnName(sim, j))
    s += "\n"
    for i in range(0,sim.rows):
        for j in range(0, sim.cols):
            col = str(tc_getColumnName(sim, j))
            if (j == 0):
                s += str(tc_getMatrixValue(sim, i, j))
```

```

else:
    p = tc_find(col)
    if tc_isA(p, "mRNA") == 1: #only output mRNA data
        s += "\t" + str(tc_getMatrixValue(sim, i, j))
    s += "\n"
    filename = str(tc_getRowName(rbs_params, n)) +
"_5x_down.tab"
    FILE = open(filename, "w")
    FILE.write(s)
    FILE.close()
    tc_setMatrixValue(rbs_params, n, 0, x)
tc_showProgress("mRNA Perturbation", int(100))
#open files
for n in range(0, rbs_params.rows):
    filename = str(tc_getRowName(rbs_params, n)) +
"_5x_down.tab"
    tc_openUrl(filename)

```

## Testing for non-monotonic input/output response (using Octave)

```

target = [0 0.3 1 0.3 0]';
inputs = [0 0.1 0.3 0.5 1]';
outputs = zeros(5,1);
ic = tinkercell.tc_getInitialValues(tinkercell.tc_allItems());
k = tinkercell.tc_getRowIndex(ic, "INPUT");
for i = 1:5
    x = inputs(i);
    tinkercell.tc_setMatrixValue(ic, k, 0, x);

```

```

tinkercell.tc_updateParameters(ic);
ss = tinkercell.tc_getSteadyState();
j = tinkercell.tc_getRowIndex(ss, "OUTPUT");
if (j > -1)
    outputs(i) = tinkercell.tc_getMatrixValue(ss, j, 0);
end
end
m = [ inputs outputs ];
m2 = toTC(m);
tinkercell.tc_plot(m2, "input-output");
score = corrcoef( target, outputs )

```

## Optimize for non-monotonic input/output response (using *python*)

```

from tinkercell import *
from tc2py import *
from numpy import *
from PSO import * #particle swarm optimization (included with
TinkerCell)

nameOfInput = "INPUT" #name of input variable in the model
nameOfOutput = "OUTPUT" #name of output variable in the model

params = tc_createMatrix(1,1)
tc_setRowName(params,0,nameOfInput)
def Objective():
    tc_setMatrixValue(params, 0, 0, 0.1)
    tc_updateParameters(params)
    ss = tc_getSteadyState()
    i = tc_getRowIndex(ss, nameOfOutput)
    x1 = tc_getMatrixValue(ss, i, 0)

```

```

tc_setMatrixValue(params, 0, 0, 2)
tc_updateParameters(params)
ss = tc_getSteadyState()
x2 = tc_getMatrixValue(ss, i, 0)
tc_setMatrixValue(params, 0, 0, 5)
tc_updateParameters(params)
ss = tc_getSteadyState()
x3 = tc_getMatrixValue(ss, i, 0)
if x3 > x1: x1 = x3
return (x2 - x1)

```

```

optimizer = ParticleSwarm()
#minimize or maximize?
optimizer.numpoints = 50
optimizer.maxiter = 10
optimizer.minimize = False
optimizer.title = "Nonmonotonic test"
g = optimizer.run(Objective,10)

```

## Create input dialogs for calling custom functions (using *python*)

```

from tinkercell import *

#callback function
def myFunc(w,h,output):
    print "width = " + str(w) + " height = " + str(h) + " output = " +
    output

#create the input window with 3 rows and 1 column
inputWindow = tc_createMatrix( 3, 1 )
tc_setMatrixValue(inputWindow, 0, 0, 0)

```



```

tc_setMatrixValue(inputWindow, 1, 0, 0.0)
tc_setMatrixValue(inputWindow, 2, 0, 0.0)

#given row names to display
tc_setRowName(inputWindow, 0, "Width")
tc_setRowName(inputWindow, 1, "Height")
tc_setRowName(inputWindow, 2, "Output")
tc_createInputWindowForScript(inputWindow, "Screenshot",
"myFunc")

#make the last row a set of options
list = ["Wiki code","HTML code"]
tc_addInputWindowOptions("Screenshot", 2, 0, toTC(list))

```

Here is the screenshot. When the user clicks the ok button, the callback function is called with the arguments 55, 12, "Wiki code"

## **Perturb all parameters with a specific prefix (using *python*)**

```

# we are just going to change all the parameters that begin with
the phrase 'synthconst'

items = tc_allItems()
params = tc_getParameters(items)

p = "" #parameter name

```

```

#just count how many there are

total = 0
for i in range(0,params.rows):
    p = tc_getRowName(params, i)
    if p.count('synthconst') > 0:
        total += 1

#for each parameter
j = 0
for i in range(0,params.rows):
    p = tc_getRowName(params, i)
    if p.count('synthconst') > 0: #check parameter name
        tc_showProgress("steady state perturbation", int((100.0 * j)/
total)) #progress meter
        j += 1
        #perturb
        s = ""
        FILE = open(p + '.perturb.txt','w+') #save to file
        p0 = tc_getMatrixValue(params, i, 0) #original parameter
value
        for q in [100, 10, 2, 1]: #perturbations
            tc_setMatrixValue(params, i, 0, p0/q)
            tc_setParameters(params,0)
            m = tc_getSteadyState() #steady state
            s += str(p0/q)
            for i in range(0,m.rows): #for each output value
                if tc_getRowName(m,i).count('m') > 0: #if name starts
with m
                    s += "\t"
                    s += str(tc_getMatrixValue(m, i, 0))
                    s += "\n"
            FILE.write(s)
            FILE.close()

#done

```

```
tc_showProgress("steady state automation", 100) #close  
progress meter
```

## Set global parameters (using *python* or *octave*)

```
#for octave add tinkercell. in front of all the tc_ commands  
p = tc_find("_") #get global item  
tc_setParameter(p,"k1" ,0.00022)  
tc_setParameter(p,"k2" ,0.19)  
tc_setParameter(p,"k3" , 0.2)  
tc_setParameter(p,"k4" , 0)  
tc_setParameter(p,"k5" , 0.0015)  
tc_setParameter(p,"k6" , 0.2)  
tc_setParameter(p,"k7" , 0.005)  
tc_setParameter(p,"k8" , 0.0001)  
tc_setParameter(p,"k9" , 0.005)  
tc_setParameter(p,"k10" , 0.0001)  
tc_setParameter(p,"k11" , 2e-06)  
tc_setParameter(p,"k12" , 0.05)  
tc_setParameter(p,"k13" , 4.5e-06)  
tc_setParameter(p,"k14" , 4e-05)  
tc_setParameter(p,"k11n" , 0.0005)  
tc_setParameter(p,"k13n" , 5e-05)  
tc_setParameter(p,"kk" , 5000)  
tc_setParameter(p,"ks" , 833)  
tc_setParameter(p,"n" , 2)  
tc_setParameter(p,"p" , 5)
```