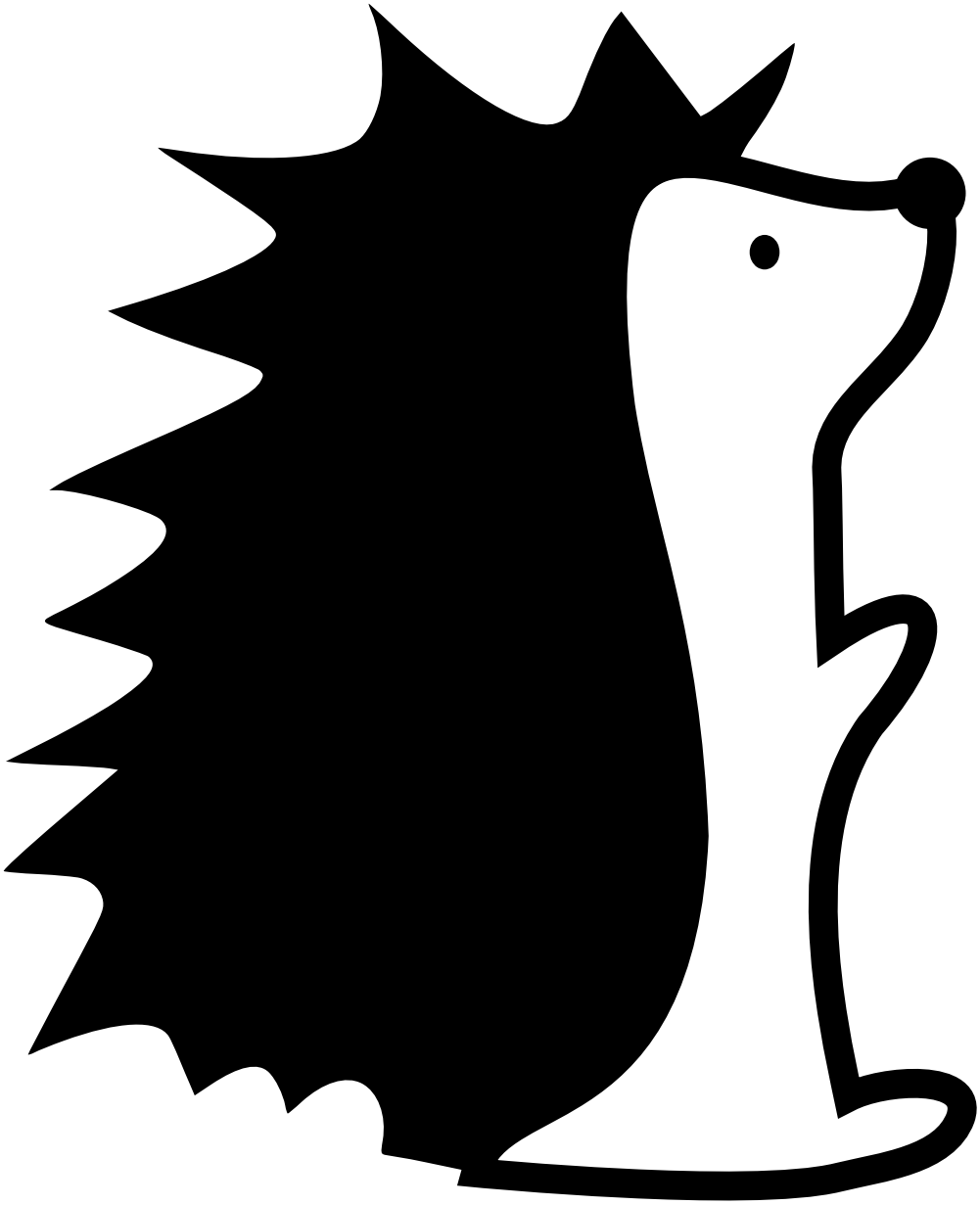


Kirpi User's Guide

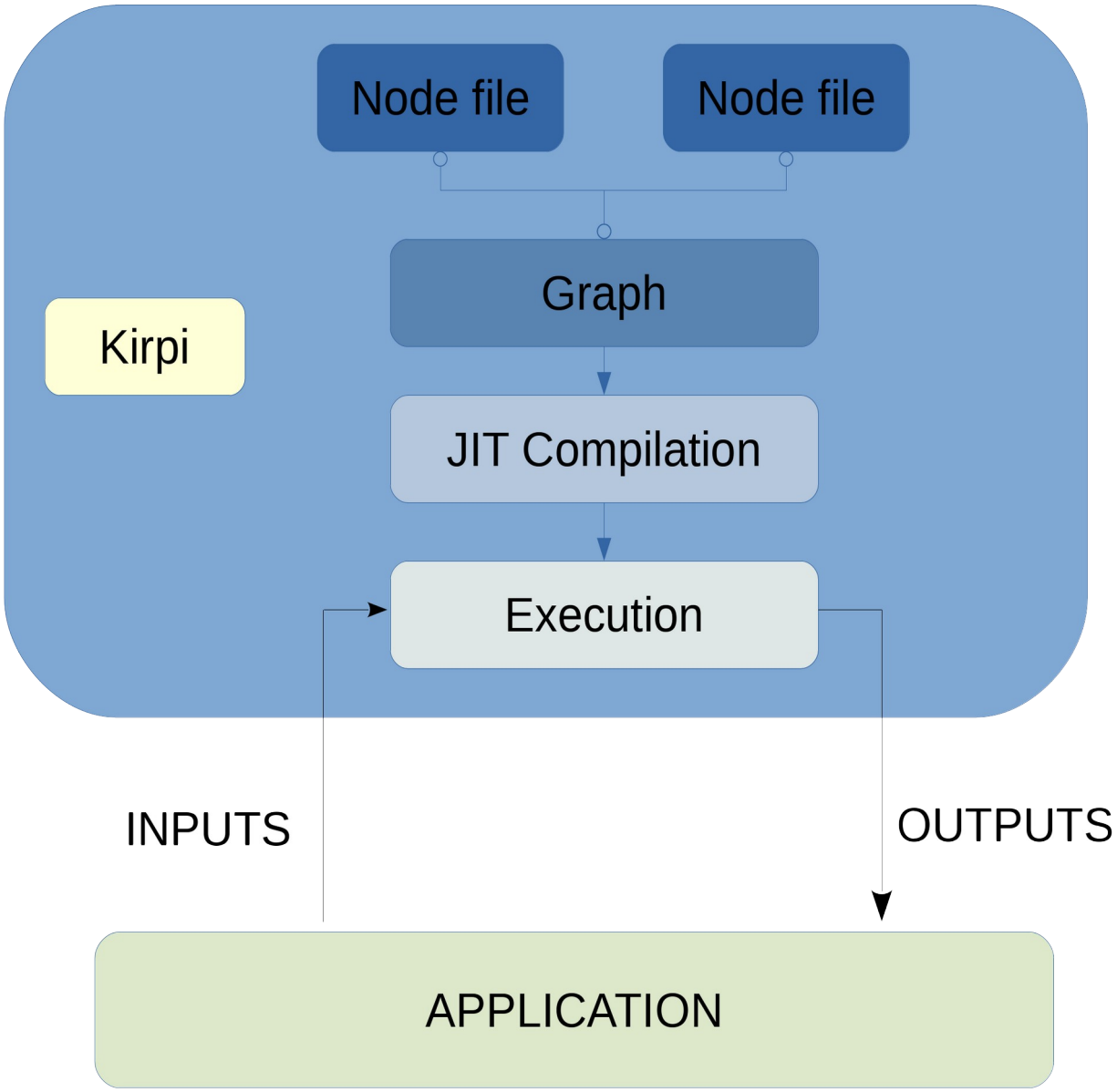
Table of Contents

1 Introduction

Kirpi makes it possible to define a process as a graph of tasks to be performed, where each task contains C++ code that is compiled and optimized on the fly.

Kirpi can be used by different user profiles:

- Non-technical users who create graphs from existing nodes.
- Technical users who use existing nodes but can also create nodes for their specific needs.
- Development teams who integrate Kirpi into their software solutions, to provide their users with tools to create specific tasks for quick prototyping and visual programming.



Kirpi can be integrated into any C++ or Python application (as a module), making it possible for developers to add functionality to the application in the form of graphs that can be viewed and edited by users, and for end users to add their own functionality (plug-ins).

Graphs can be predefined and executed as is, or they can be defined dynamically by loading different graphs and by editing them interactively.

Graphs are composed of basic, predefined nodes, or of nodes created by technical users or development teams. Nodes are written in C++, must adhere to the C++14 standard, and can use header files and modules provided by any other library.

Kirpi is a visual programming framework:

- Quick prototyping for R&D teams.
- Simplified creation of plug-ins.
- Code customization for advanced users.
- Integration in any C++ project.
- Generation of standalone applications.

The advantages of Kirpi:

- Just-in-time compilation transparent to the user.
- Automatic graph optimization.
- Vectorized execution, separating algorithm from parallelization.
- Integrated profiling.
- Nodes written in C++:
  - Well-known language.
  - No need to provide bindings for other languages.
  - Tools to generate nodes automatically from existing libraries.
  - Maximum performance.
  - Use of standard debugging tools for development teams.

The remainder of this user's guide consists of:

- An explanation of how to install Kirpi and what the installation contains.
- Hands-on tutorials for getting acquainted with Kirpi.
- A high-level guide to using Kirpi, which complements the detailed documentation provided by the Kirpi libraries and applications.
- More complex examples of what can be done with Kirpi.
- Frequently asked questions (FAQ) about Kirpi.

2 Installing Kirpi

This section explains how to install Kirpi and what the Kirpi installation contains. Kirpi can be installed on Linux or on Windows.

2.1 Linux Prerequisites

On Linux, Kirpi runs on CentOS 7 (versions 7.4 and later). Kirpi may well work on other Linux distributions as well, but it has only been tested on CentOS, and for now the installer will refuse to install Kirpi on other Linux distributions.

Kirpi 2020 requires that gcc 6.3.1 be installed on the system; Kirpi 2022 requires gcc 9.3.1. Both versions require that C++ support and the standard C++ libraries be installed, as well as libcurl. If that is not the case, the Kirpi installer will propose to install them for you. Note that installing gcc and libcurl requires super-user privileges, but installing Kirpi itself does not.

2.2 Windows Prerequisites

Kirpi has been tested on Windows 7 and 10, and should run on those or any more recent versions of Windows.

Kirpi requires that the Visual C++ runtime libraries be installed on the system. Those libraries are installed with Visual C++ 2015, 2017, 2019 and 2022. If they are not present, the Kirpi installer will propose to install them for you. Note that installing the C++ runtime libraries requires administrator privileges, but installing Kirpi itself does not.

In order to compile graphs, Kirpi also requires that Visual Studio (2019 or later) with C++ support be installed. If Visual C++ support is not installed, you will still be able to use Kirpi to execute graphs which have been compiled elsewhere, but you will not be able to compile them yourself.

2.3 Which Version to Install?

Two versions of Kirpi are available for installation: Kirpi 2020 and Kirpi 2022. In general you should prefer Kirpi 2022 unless you need to build plug-ins compatible with Maya 2020 (plug-ins built with Kirpi 2022 are compatible with Maya 2022). You can also install both versions, if you so desire (in two different locations).

2.4 Installation Procedure

Kirpi's interactive installer can be run from the command line or by double-clicking on it, and does the following:

1. Checks whether required system dependencies have been installed, and proposes to install them if not.
2. Downloads the necessary Kirpi components from the Caleido-scop website.

3. Installs Kirpi in a directory chosen by the user.
4. *[Windows only]* Creates shortcuts for the desktop and Start Menu.

The installer must be run with super-user/administrator privileges to install missing system dependencies, but not to install Kirpi itself, which can be installed in any directory.

> **Notes for Windows**
>
> The Windows installer asks whether Kirpi is to be installed for the current user only or for all users. In the latter case, which requires administrator privileges, the desktop and Start Menu shortcuts are made available to all users of the machine (that is the only difference).
>
> Note that Kirpi must **not** be installed to the default Windows program location, *e.g.* `C:\Program Files`. Kirpi writes bitcode files for compiled graph nodes as needed, and Windows does not allow applications to write in `C:\Program Files`. So by default, Kirpi is installed to `C:\kirpi`, although you are free to specify a different location.

The installer can also be run without the graphical user interface, in "headless" mode: run `KirpiInstaller --help` for details.

Once the installer has finished, the installation directory will contain a `kirpi` subdirectory containing Kirpi itself (the contents are described below), as well as a program called `maintenancetool` which can be used to:

- update Kirpi
- add or remove Kirpi components, such as documentation, examples, tutorials and plug-ins
- remove Kirpi entirely

The installer does **not** add Kirpi's `bin` folder to the PATH environment variable, so you may wish to do so yourself: see Setting Up the Kirpi Environment below.

2.5 Installation Contents

The `kirpi` directory created by the installer contains the following subdirectories.

`lib` and `include` contain the libraries and header files provided by Kirpi (*libkirpi*, *graphview*, *kpui* and *kpmaya*). See Kirpi Libraries below for more information about each.

`bin` contains a few standalone programs provided by Kirpi (*kpexec*, *kpedit* and *nodegen*) as well as DLLs for Windows. See Kirpi Applications below for more information about each.

`doc` contains this user's guide, as well as documentation for the *kpexec* and *nodegen* applications, and Doxygen-based HTML documentation for each of the Kirpi libraries.

`nodes` contains a number of predefined graph nodes that users might find helpful, either for use as is or as models for creating new nodes.

`presets` contains graph node presets that may come in handy for users creating new graph nodes (when editing nodes in *kpedit*, you can load one or more presets containing appropriate compiler options, among other things).

`tutorials` and `examples` contain a number of graphs, source code and other files referenced by the tutorials and examples below.

`maya` contains several examples of Maya plug-ins (precompiled plug-ins and supporting MEL scripts). For details about the plug-ins, see the Examples section below.

Finally, Kirpi comes with a number of free, open-source libraries used by Kirpi or by the tutorials and examples, such as LLVM/Clang, Qt, TBB, Eigen, pybind11 and OpenImageIO. Licensing information for those libraries can be found in the corresponding directories (for LLVM, Qt, Eigen and pybind11) or in the `licenses` directory (for all the others).

Kirpi Libraries

Header files and precompiled library files for each of the following libraries can be found in the Kirpi installation's `include` and `lib` directories, respectively. Doxygen-based documentation for each can be found in the `doc` directory.

To compile, link and run your own programs using the Kirpi libraries:

- compile with `-IKIRPI_ROOT_DIR/include`
- link with `-LKIRPI_ROOT_DIR/lib`
- to run, add `KIRPI_ROOT_DIR/lib` to the LD_LIBRARY_PATH environment variable on Linux, or add `KIRPI_ROOT_DIR\bin` to the PATH environment variable on Windows

where KIRPI_ROOT_DIR is the path of the `kirpi` directory created by the installer.

libkirpi

libkirpi is a low-level library that allows you to define a Kirpi graph, where each node is essentially a C++ function with any number of inputs or outputs. It uses LLVM and Clang to compile the graph in memory, to execute it, to set its inputs and retrieve its outputs.

graphview

graphview is a library containing Qt classes for viewing and editing graphs. It provides specialized subclasses of QGraphicsView and QGraphicsScene that can be used with any sort of graph that implements its abstract model.

kpui

The *kpui* library is an extension of *graphview* classes for viewing and editing Kirpi graphs specifically. It provides a Qt widget for editing graphs, and dialog boxes for creating new graphs and modifying existing ones.

kpmaya

The *kpmaya* library defines some classes that can simplify the use of Kirpi graphs in Maya plug-ins.

Kirpi Applications

kpexec

kpexec is a simple command-line application that allows you to load a graph, compile it, set its inputs, execute it and retrieve its outputs. The source code for *kpexec* can be found in the `examples` directory; documentation can be found in `doc/kpexec.rst`. The source code is also explained in some detail in the Examples section below.

kpedit

kpedit is a graphical application that allows you to view, edit and execute a Kirpi graph. For tips and detailed instructions about using *kpedit*, see the tutorials and the Using Kirpi section below.

nodegen

nodegen is a command-line application that can be used to generate Kirpi graph nodes from existing C++ code. Given the definition of a single function, it can generate a graph node containing code copied from the function. Or, given the declaration of a C++ class, it can generate a node for each method defined by the class, where nodes simply call the corresponding class methods. See `doc/nodegen.rst` for details.

2.6 Kirpi Licensing

Kirpi applications and the Kirpi API can be used by anybody, without a license, to execute graphs. But to view, edit and compile graphs, you must have a Kirpi license.

Caleido-scop provides licensed Kirpi users with a file called `kirpi.lic` which should be placed in the `kirpi` directory created by the installer. If you need to put the license file elsewhere, or give it a different name, you can do so: just set the environment variable KIRPI_LICENSE_FILE to the path of the file.

2.7 Setting Up the Kirpi Environment

All Kirpi applications are installed to the `bin` folder created by the installer (as are DLLs on Windows). As mentioned above, the `bin` folder is not added automatically to the PATH environment variable, so you may wish to do so yourself.

Any application or plug-in using Kirpi needs for a couple of environment variables to be set beforehand: KIRPI_ROOT_DIR must be set to the path where Kirpi was installed; and LD_LIBRARY_PATH (on Linux) or PATH (on Windows) must point to the directories where Kirpi's shared libraries are installed.

For applications installed with Kirpi, such as *kpedit*, the environment is set up automatically: the `bin` folder contains a shell script for each, which sets up the environment before running the actual application. On Linux, `kpedit` is actually a shell script; on Windows, `bin` contains a batch file called `kpedit.bat`. So you can run those applications without setting any environment variables beforehand.

If, however, you are going to run a third-party program such as Maya, or run your own Kirpi-based programs, you will need to set up the environment explicitly using one of the scripts provided by Kirpi:

- `kpenv.sh` (for sh-based shells such as bash)
- `kpenv.csh` (for csh-based shells such as tcsh)
- `kpenv.bat` (for Windows)

3 Tutorials

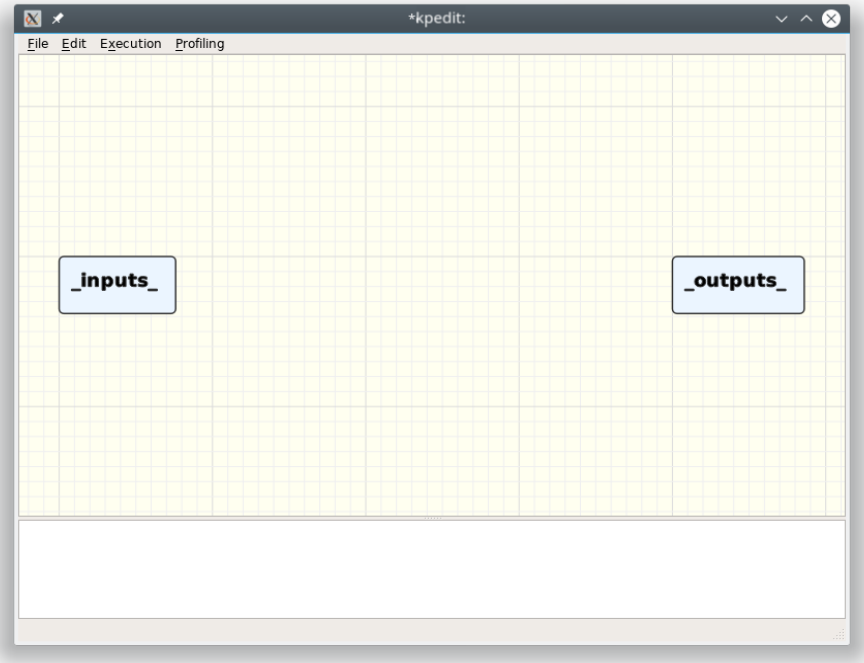
The following tutorials are very simple examples whose only purpose is to demonstrate how a graph works and how to create graph nodes. These examples perform very simple operations but are not particularly useful. In the Examples chapter you will find more realistic uses, both standalone and integrated into other applications.

The graph files referenced in the tutorials can be found in the Kirpi installation's `tutorials` directory.

Tutorial 0: Adding Two Numbers

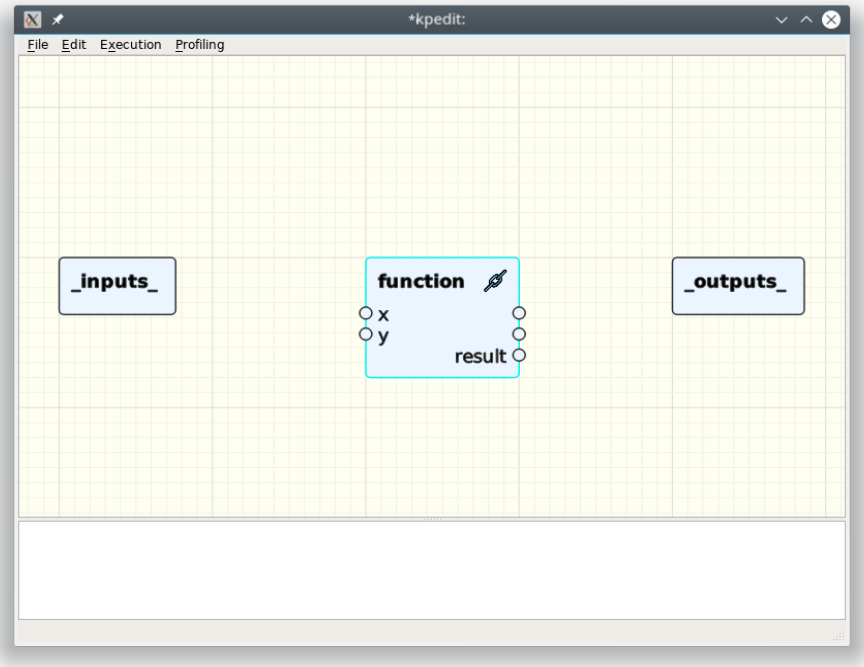
The purpose of this tutorial is to demonstrate the fundamentals of Kirpi by simply adding two numbers and displaying the result.

To begin, run the *kpedit* application in a shell (or by double-clicking on its shortcut in Windows).



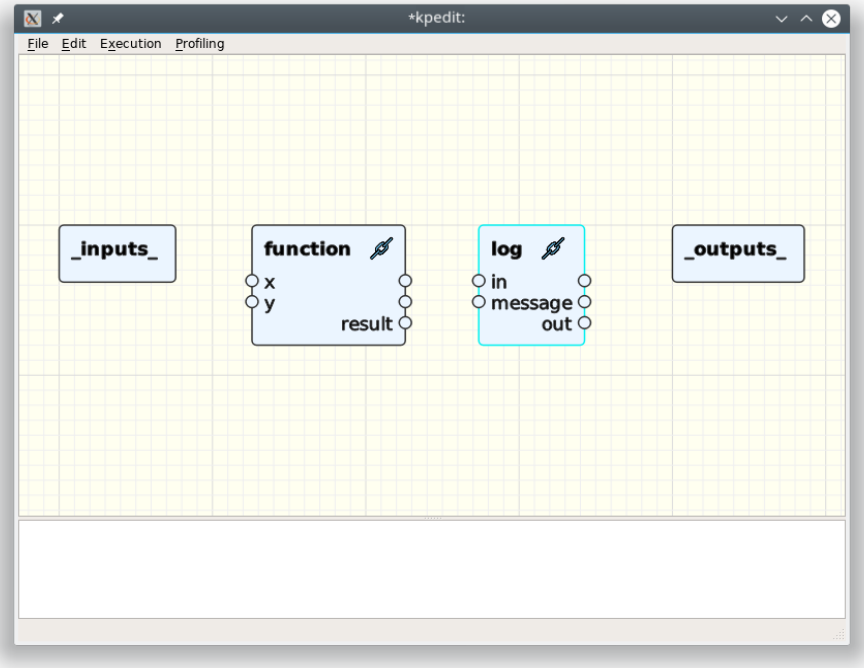
The initial graph contains two nodes already, defining the graph's inputs and outputs. (A Kirpi graph always contains an *_inputs_* node and an *_outputs_* node.)

Now add a node called *function* to the graph: click with the right mouse button and select *Load Node → tutorial_nodes → function* from the popup menu. Alternatively, you can press the tab key, start typing the name "function" in the widget that appears and select the node from the popup list. Or you can click with the right mouse button, select *Browse Nodes…* from the popup menu, click on *tutorial_nodes* in the lefthand pane of the dialog box that appears, then select *function.node* in the righthand pane. These are all equivalent ways of loading the file `function.node`.

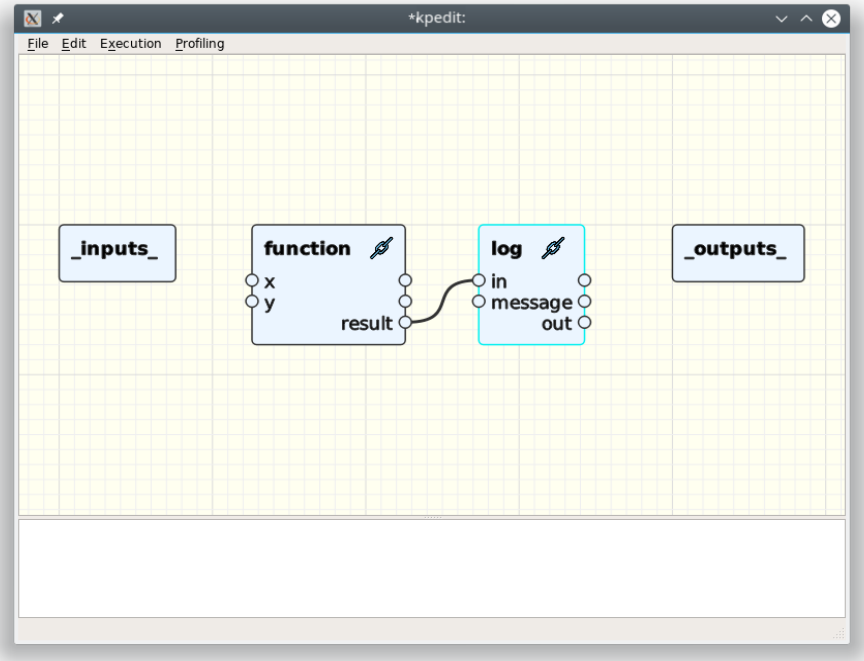


The *function* node is added to the graph at the location where the mouse was clicked. It has two inputs (*x* and *y*) and one output (*result*). The link icon in its upper right corner indicates that it references a node file on disk.

Using the same procedure, add a *log* node to the graph by selecting `log.node` from the directory `tutorial_nodes`. The *log* node has two inputs (*in* and *message*) and one output (*out*).



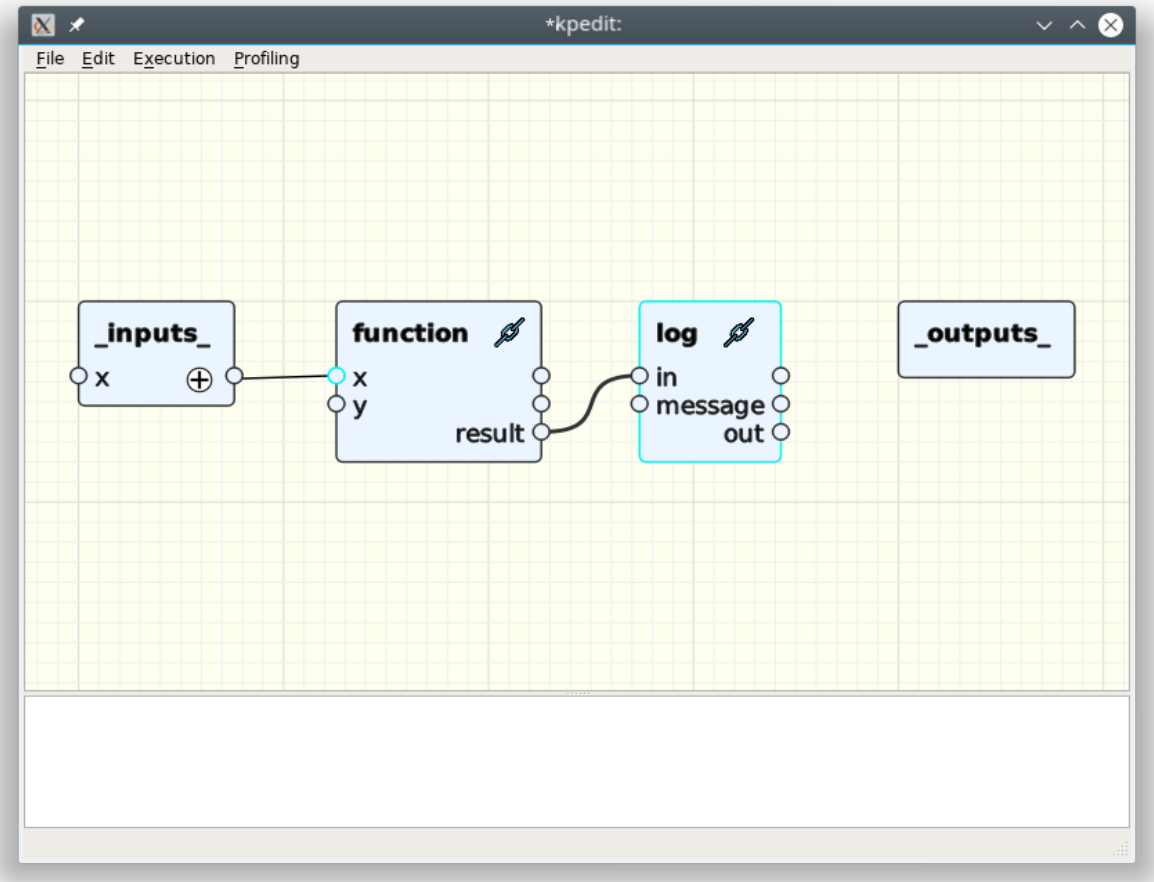
Now connect the *result* output of the *function* node to the *in* input of the *log* node. Click with the left mouse button on the circle to the right of the *result* output and, keeping the mouse button pressed, move the mouse to the circle to the left of the *in* input of the *log* node, then release the mouse button. While you are moving the mouse, a curved line appears from the starting point to the current position of the mouse.



The functionality of the graph is now complete: an addition followed by the display of the result. Two things are missing, however:

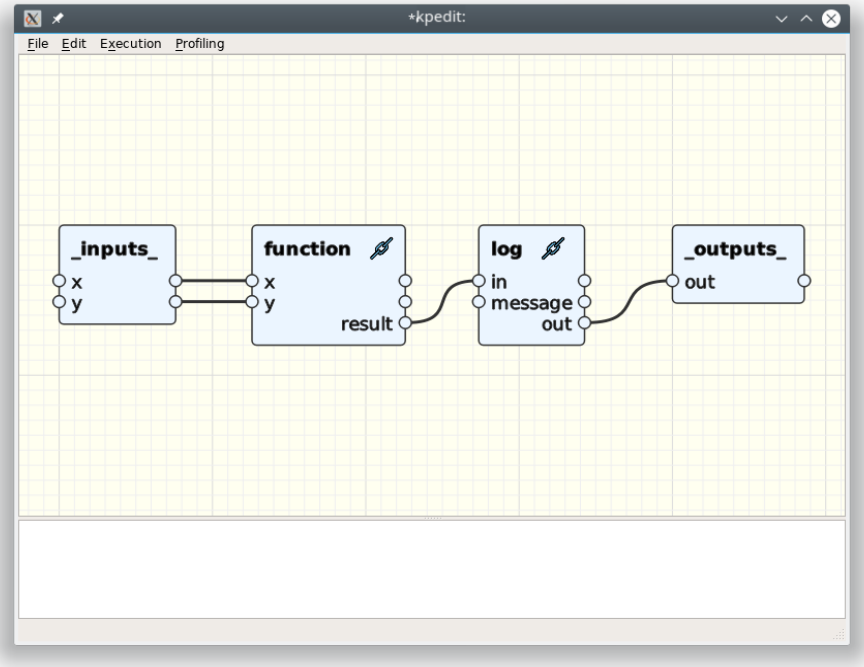
- You must provide input values to the *function* node.
- Only nodes connected to the graph's outputs are evaluated, so the *log* node must be connected to the *_outputs_* node.

To add input values to the graph, click with the left mouse button on the circle to the left of the *x* input of the *function* node, then drag the mouse over the *_inputs_* node. The mouse pointer turns into a + sign. Release the mouse button and an *x* input is automatically added to the *_inputs_* node.

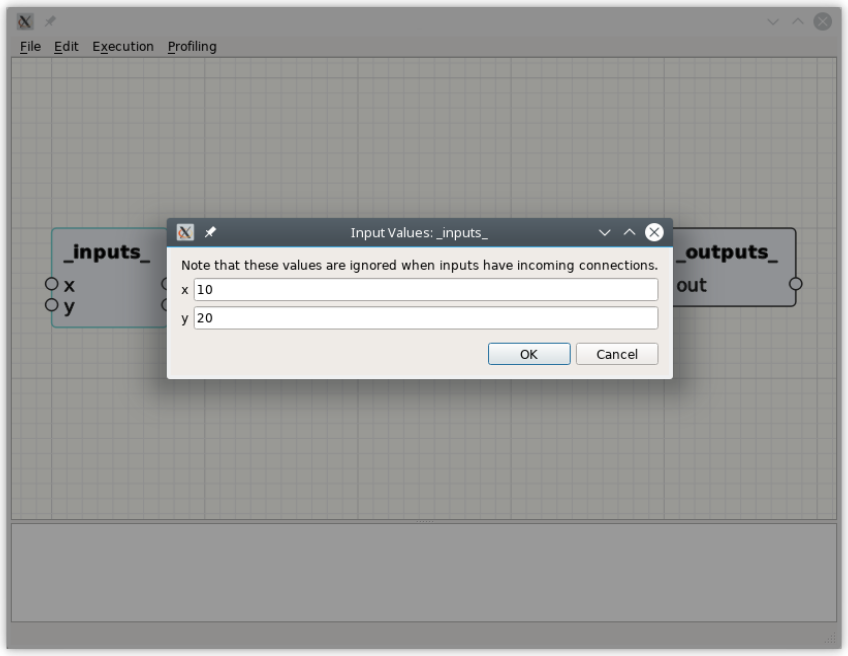


Now do the same with the *y* input of the *function* node.

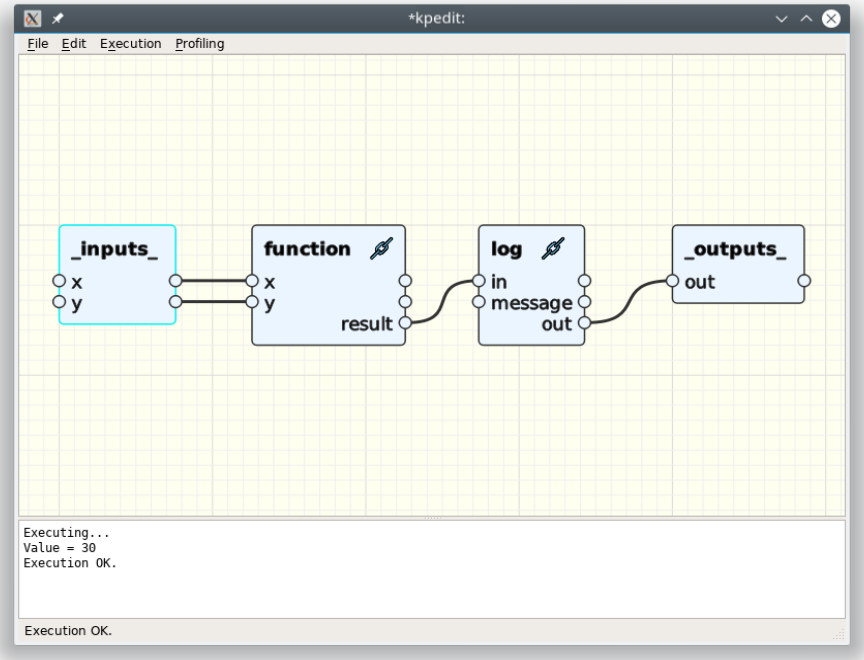
To add an output to the graph, do the same as above, but this time click on the *out* output of the *log* node and release the button over the *_outputs_* node.



Set values for the graph's inputs by clicking with the right mouse button on the *_inputs_* node and selecting *Edit Input Values* from the popup menu. In the window that appears, enter values for the *x* and *y* inputs: 10 and 20, for example.



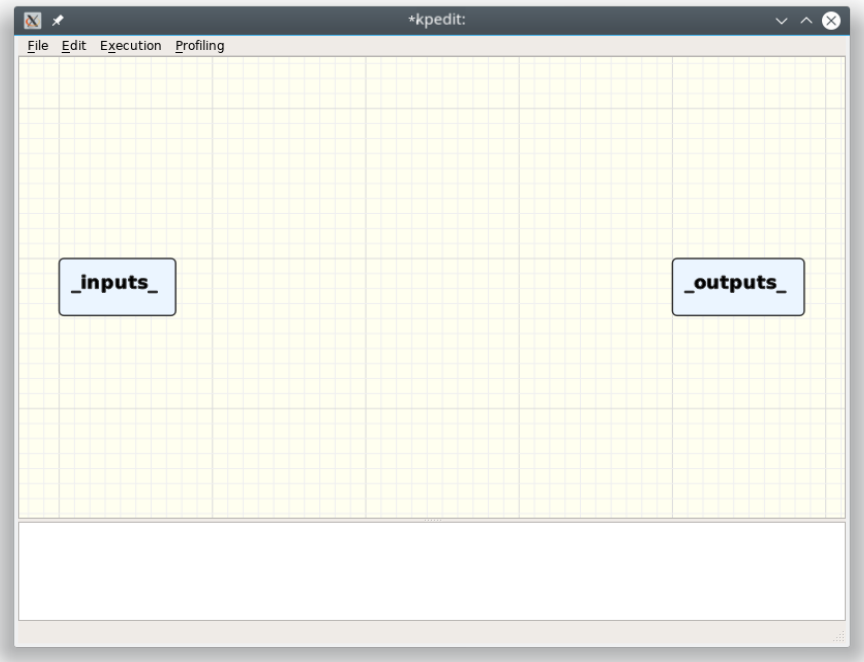
You can now execute the graph, by clicking on *Execute* under *Execution* in the menu bar.



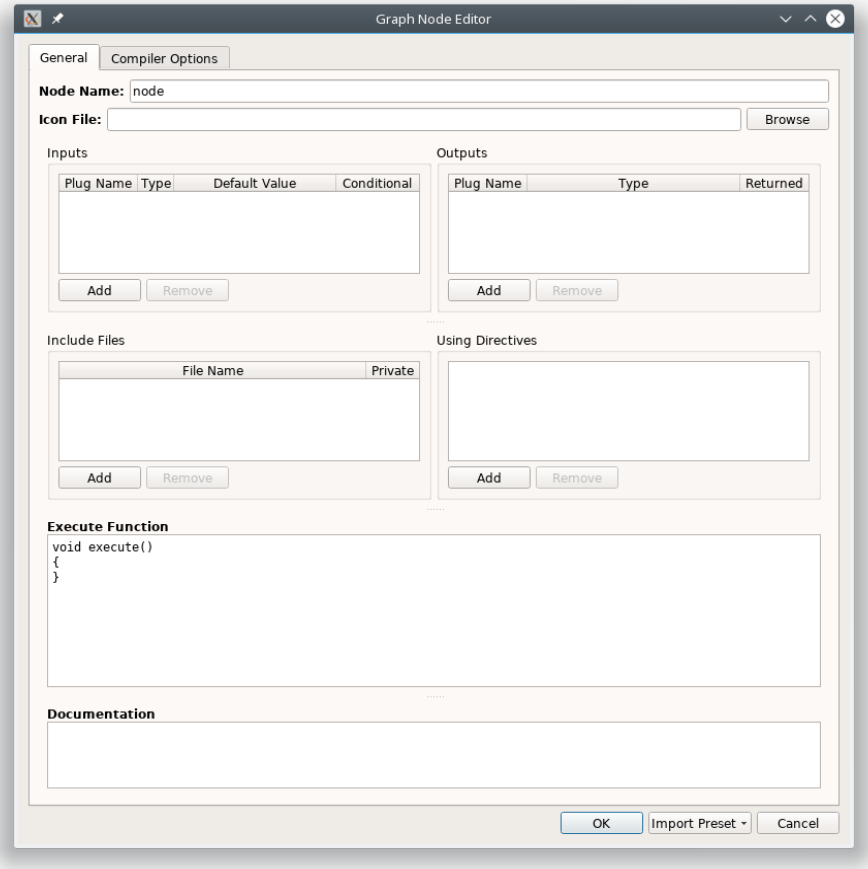
Tutorial 1: Adding Character Strings

In this tutorial we will see how to create a simple graph node.

To begin, run the *kpedit* application in a shell (or by double-clicking on its shortcut in Windows).



Click with the right mouse button in the empty space around the middle of the window, and select *Create Node* from the menu that pops up. The Graph Node Editor opens as in the figure below.



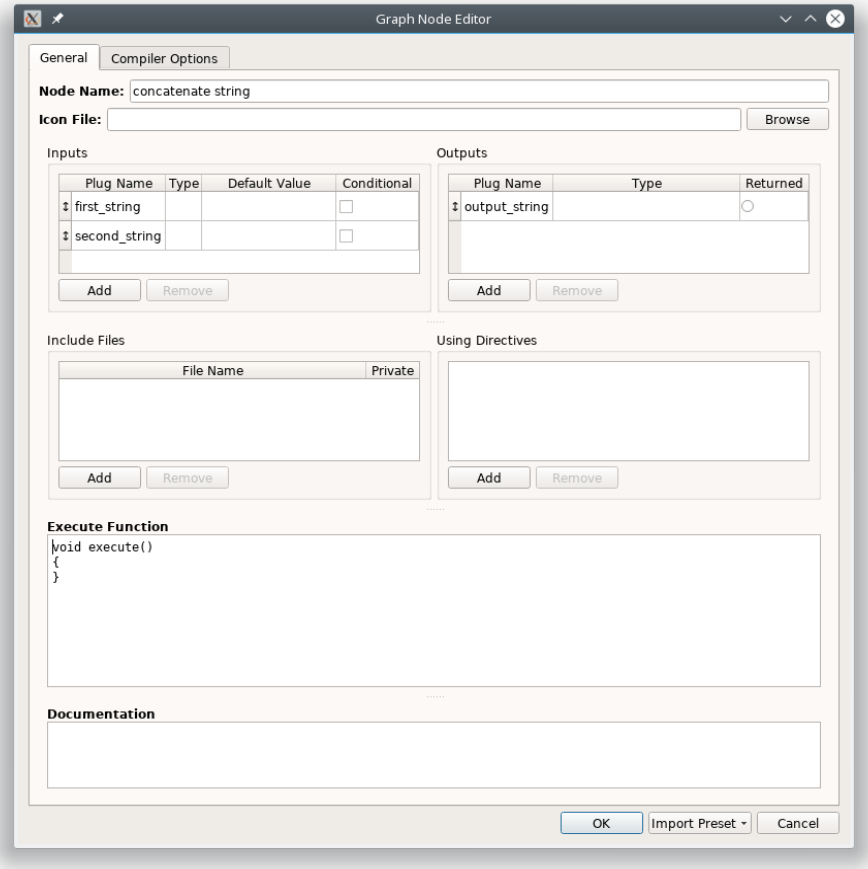
Our node will take two character strings as inputs and return the result of joining the two strings, separated by a space.

Begin by filling in the *Node Name* field, replacing the generic "node" name with "concatenate string".

Add the first input string by clicking on the *Add* button below the *Inputs* box. The *Name* field is activated automatically; enter the name "first_string".

Add a second input string the same way, calling it "second_string".

Now click on the *Add* button below the *Outputs* box to add an output named "output_string".



Now enter the node's execute function. The code looks like this:

```
void execute(const std::string& first_string,
             const std::string& second_string,
             std::string& output_string)
{
    output_string = first_string + " " + second_string;
}
```

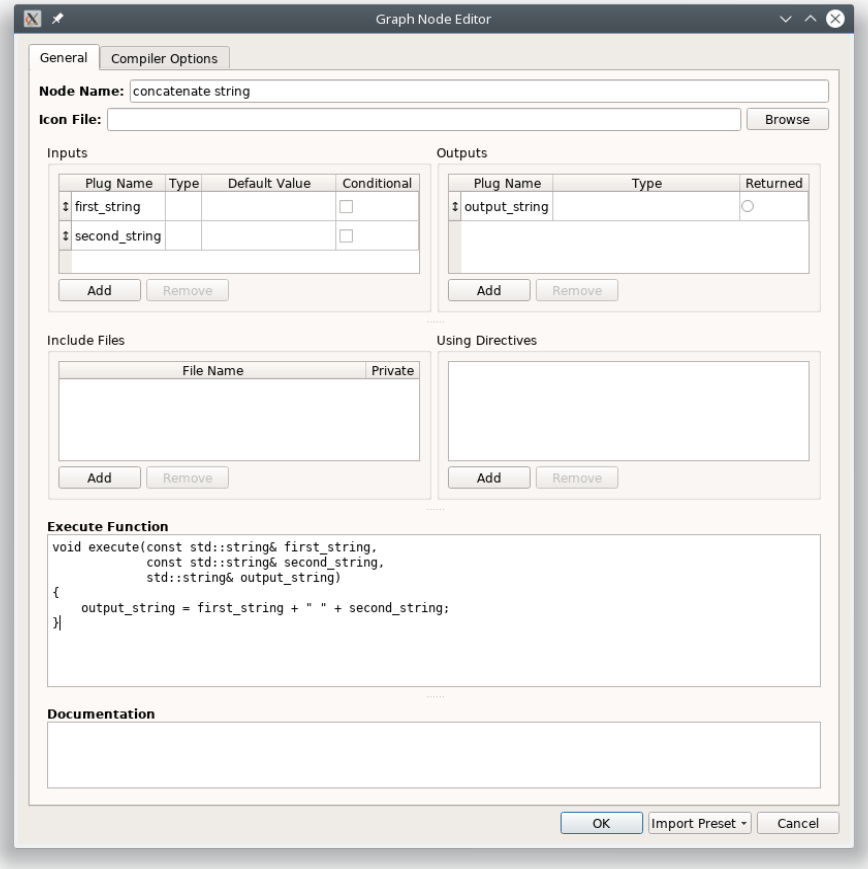
A few points worth noting:

- All nodes must have a function called *execute*, which is called when the node is executed.
- **The function's parameter names must exactly match those declared in the *Inputs* and *Outputs* boxes.** In this case, the inputs are called "first_string" and "second_string", the output is called "output_string", and the execute function's parameters have the same names.
- The execute function can also return a value, as we shall see in Tutorial 4.

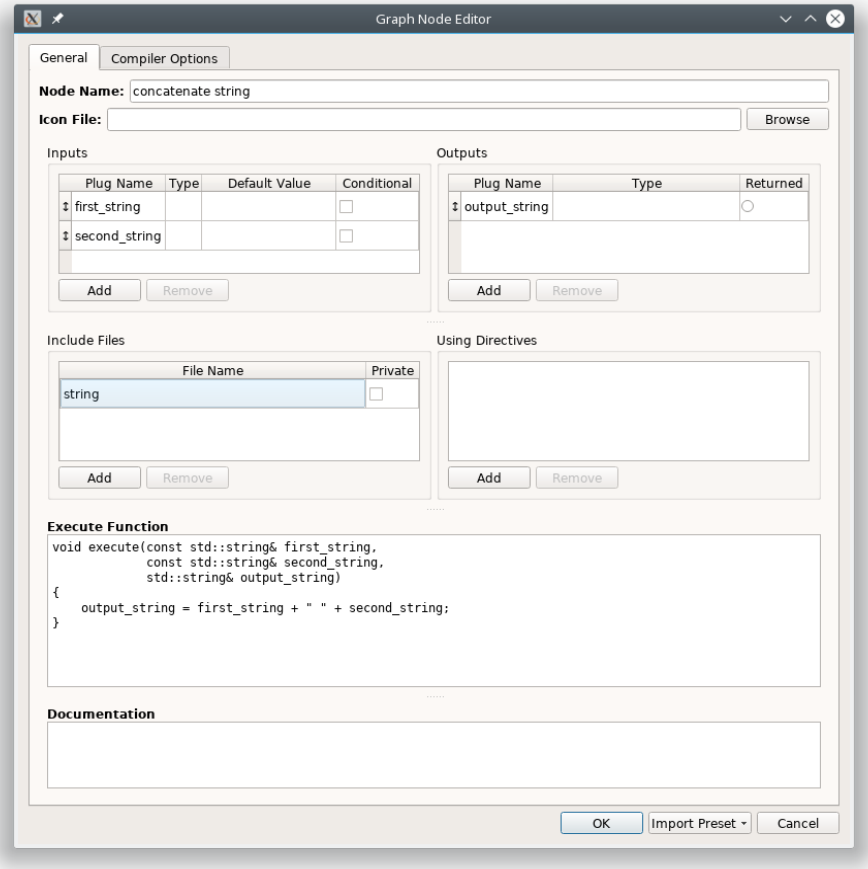
The input parameters are of type *const std::string&*, which identifies them as inputs (because they are *const*). Passing them by reference rather than by value is more efficient because it avoids making copies of them.

The output parameter is passed by non-const reference so that it can be modified by the function.

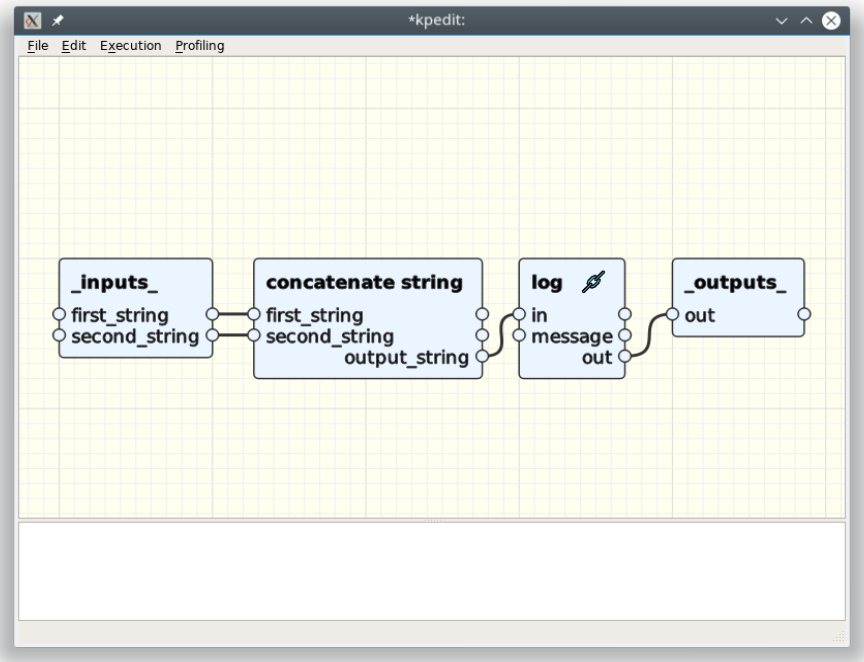
The body of the function joins the two input strings, separating them with a space.



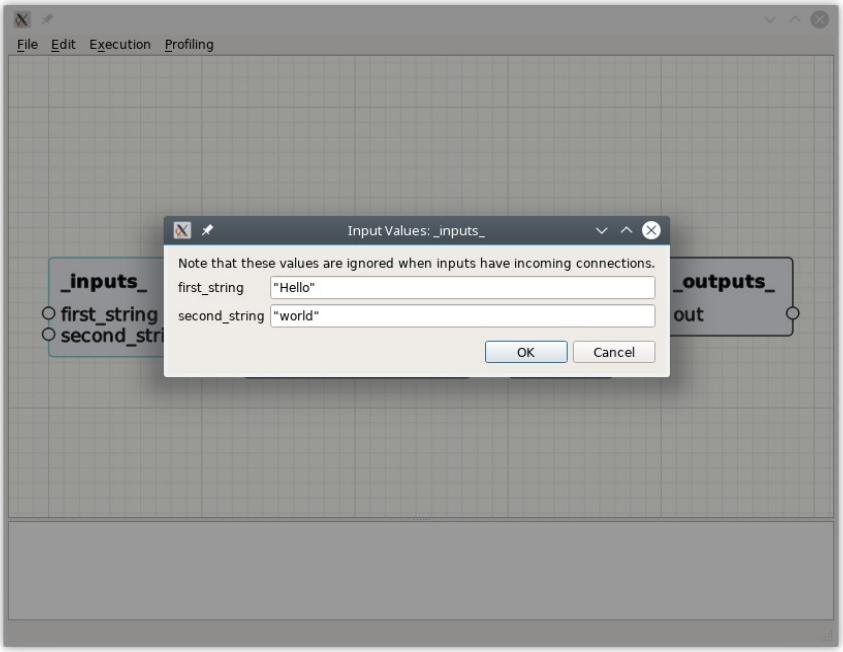
Because *std::string* is not a built-in type in C++, the node must include the header file where the type is defined. To do so, click on the *Add* button below the *Include Files* box and enter "string". This corresponds to the usual `#include <string>` line found in normal C++ code.



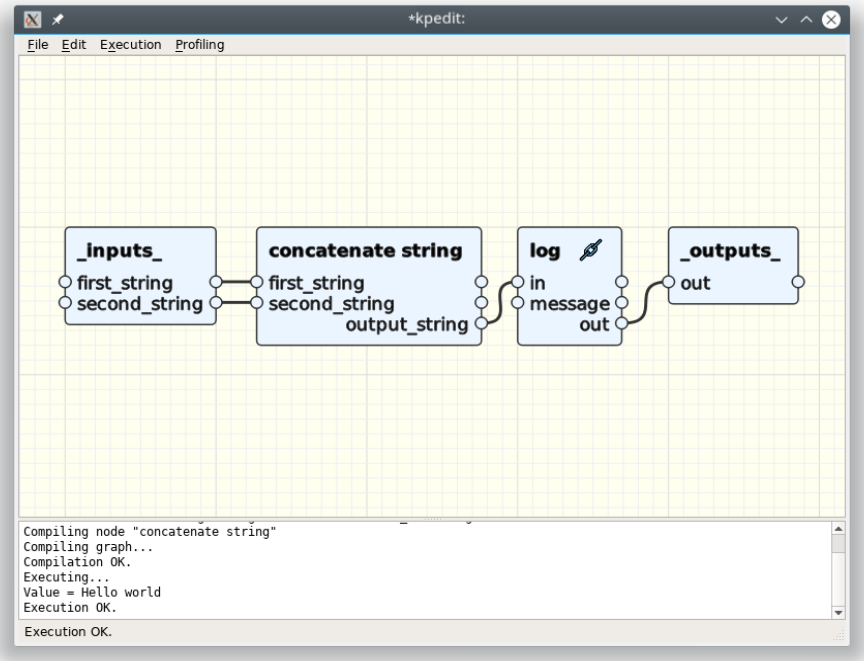
After clicking on the *OK* button, add a *log* node, then connect it as in the preceding tutorial so that you end up with a graph like the following:



Finally, set values for the graph's inputs:



Execute the graph by clicking on *Execute* under *Execution* in the menu bar:



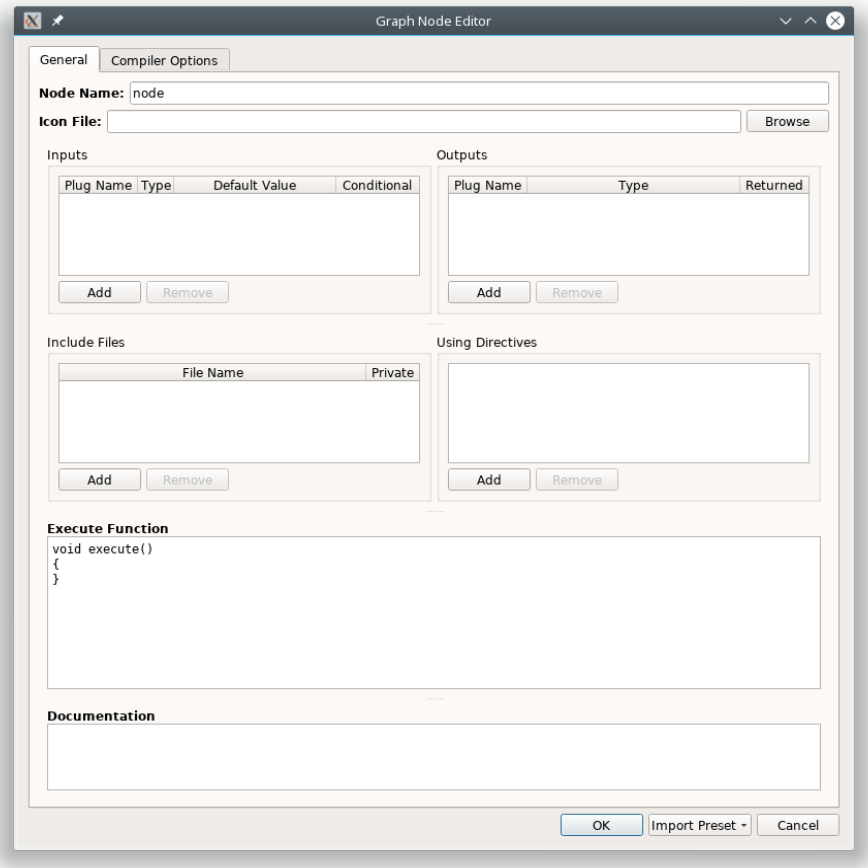
Important point to remember: when a node needs to include a header file, add it to the node's *Include Files* box.

Tutorial 2: Using Templates

Kirpi supports the use of templates in the nodes comprising a graph. In this tutorial we will see how to use a template to perform a simple addition operation that can be used on inputs of any type that implements the C++ addition operator. The resulting node can be used to add integers, strings, vectors, images, etc.

The use of templates simplifies the development but also the use of graphs, because a single node can be used to perform the same operation on many data types.

Start up *kpedit* as in the previous tutorial (or, if you are already in *kpedit*, start a new graph by selecting *File → New* in the menu bar), then create a new node by clicking with the right mouse button (anywhere but on an existing node) and selecting *Create Node* from the popup menu. The Graph Node Editor opens as shown below.

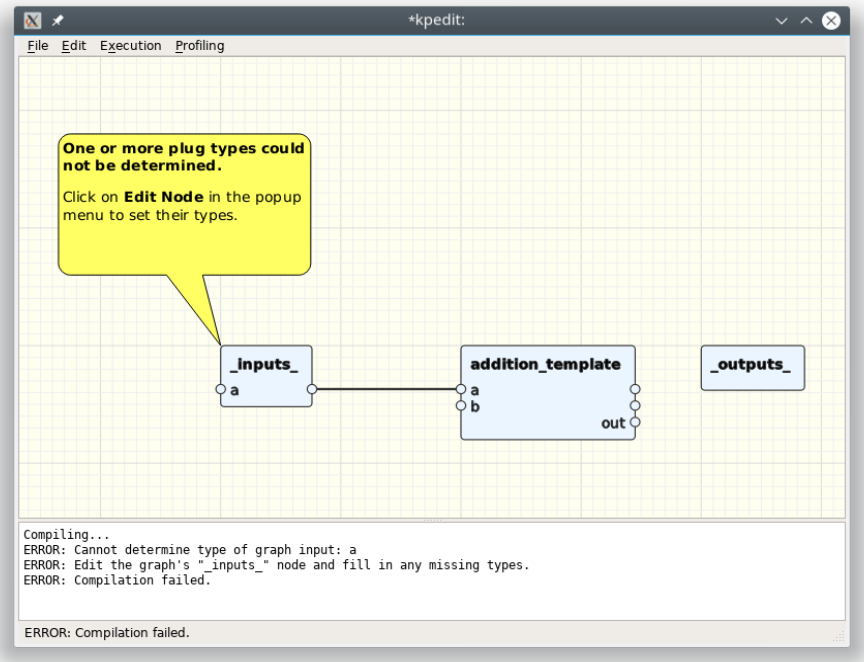


Call the node "addition_template", then create two inputs (*a* and *b*) and one output (*out*).

In the *Execute Function* box, enter the following code:

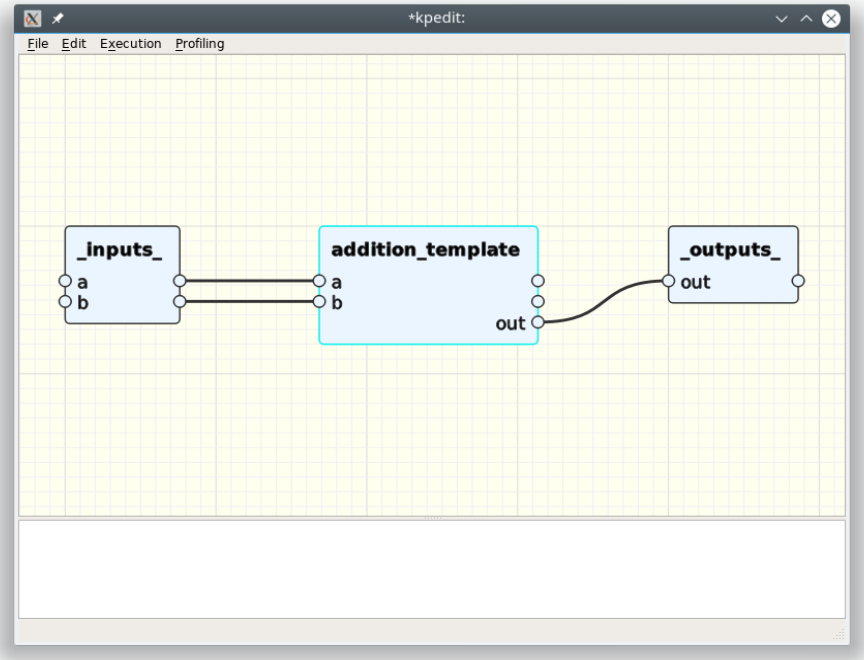
```
template<typename T>
void execute(const T& a, const T& b, T& out)
{
    out = a + b;
}
```

Now create the graph's inputs and output as in the preceding tutorials. This time, however, when you create a new input, a message appears asking you to supply a type for it:



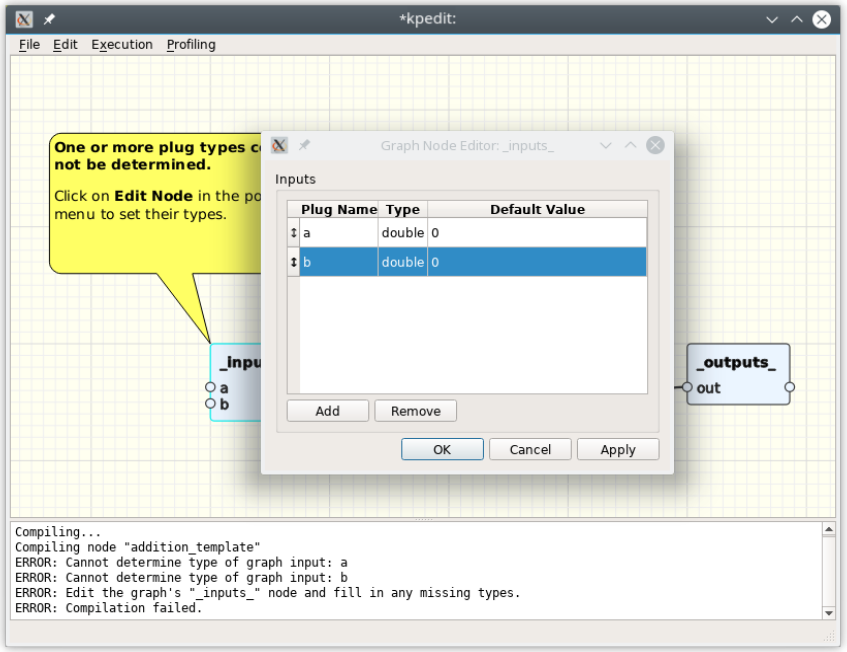
Because the template has not been instanced yet, and gives no default values, making type inference impossible, it is up to the user to supply a type for the input. If you carry on, connecting *a* and *b* to the *_inputs_* node, and *out* to the *_outputs_* node, the following messages appear in the output window:

```
ERROR: Cannot determine type of graph input: a
ERROR: Cannot determine type of graph input: b
ERROR: Edit the graph's "_inputs_" node and fill in any missing types.
ERROR: Compilation failed.
```



Because the node can take inputs of any type, and generate an output any type, and because those types have not been specified, Kirpi cannot compile the code.

To specify types for the inputs, click with the right mouse button on the *_inputs_* node, and select *Edit Node* from the popup menu.



For each input and output plug, enter "double", for example, in the *Type* column, and "0" in the *Default Value* column. Now the graph can be compiled and executed without error.

Save the node to a `.node` file so that it can be reused: click on the "addition_template" node with the right mouse button and select *Write Node to File* from the popup menu.

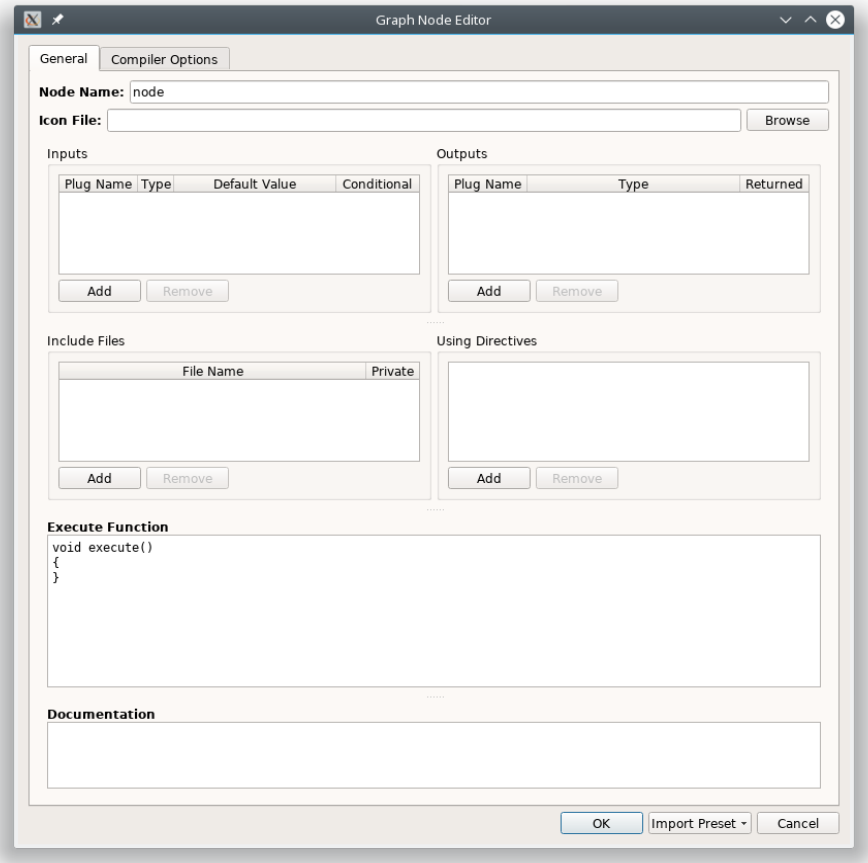
Note that if the inputs of the "addition_template" node had been connected to the outputs of other nodes whose types were known, it would be unnecessary to specify types for the node's inputs and output values: their types would be deduced automatically. Nonetheless, we recommend always specifying input and output types, even when it is not strictly necessary, to avoid errors later on. We also recommend specifying default values, for plain old data types like *int* or *float*, to avoid undefined results.

As an example, try creating a graph using a copy of the node you just created, where the inputs are not of the same type as the first node. An example graph is provided with the tutorial files.

Tutorial 3: Returning a Value

In this tutorial we will see how a node's execute function can return a value directly, rather than being passed an output parameter by reference.

As in the preceding tutorial, create a new node by clicking on the right mouse button and selecting *Create Node* from the popup menu. The Graph Node Editor opens as shown below.

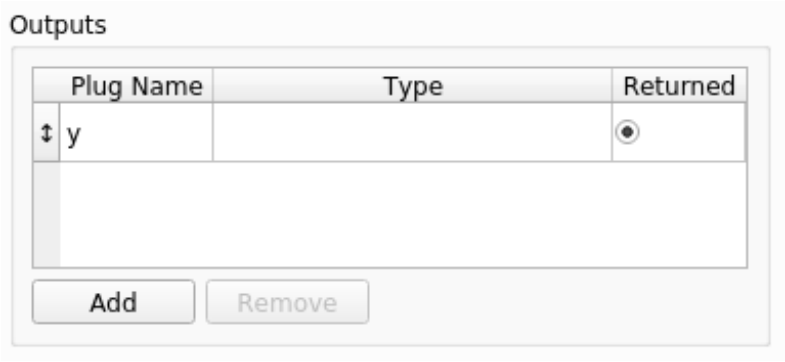


Call the node "square", and give it an input called *x* and an output called *y*.

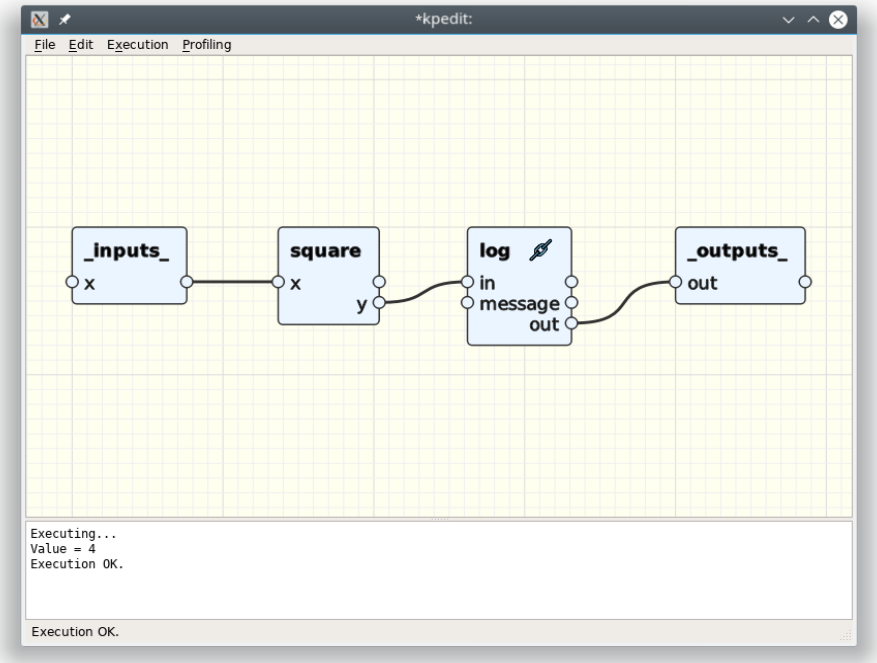
In the *Execute Function* box, enter the following code:

```
float execute(float x)
{
    return x*x;
}
```

This is just a normal function, but in the *Outputs* box you have to check the *Returned* check box corresponding to the output *y:*



This tells Kirpi that the output's value is returned directly by the function and that there is no corresponding output parameter.



If you forget to check *Returned* for the output, compilation of the graph will fail with a message like the following:

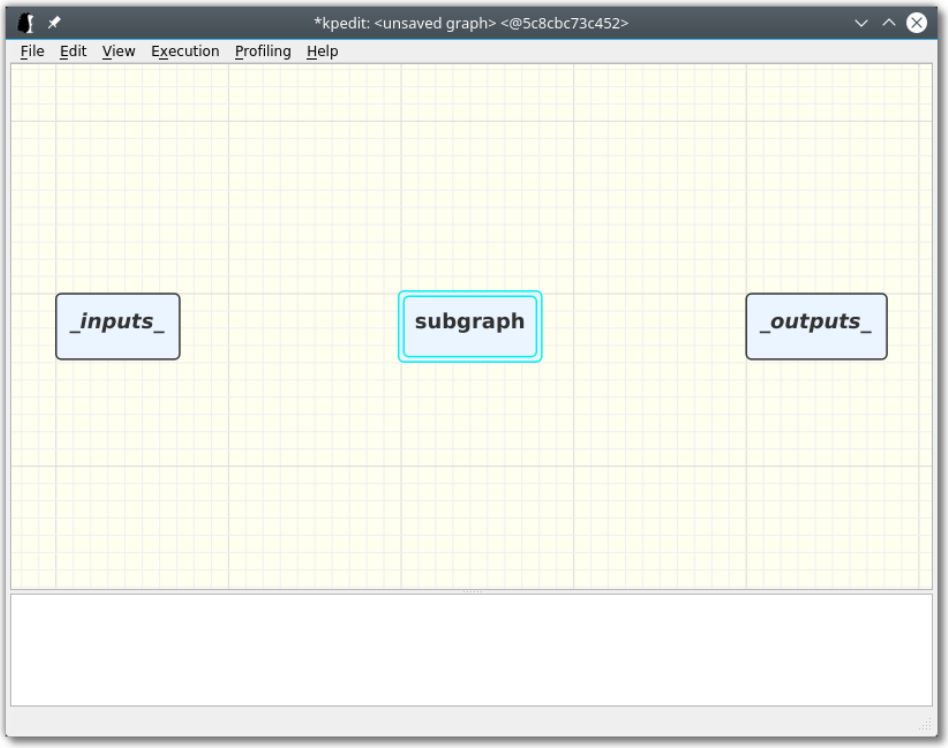
```
WARNING: Execute function's return value is not used: node "square"
ERROR: Output parameter not found: y
ERROR: One or more inputs or outputs are not declared by the execute
function: node "square"
```

Tutorial 4: Subgraphs

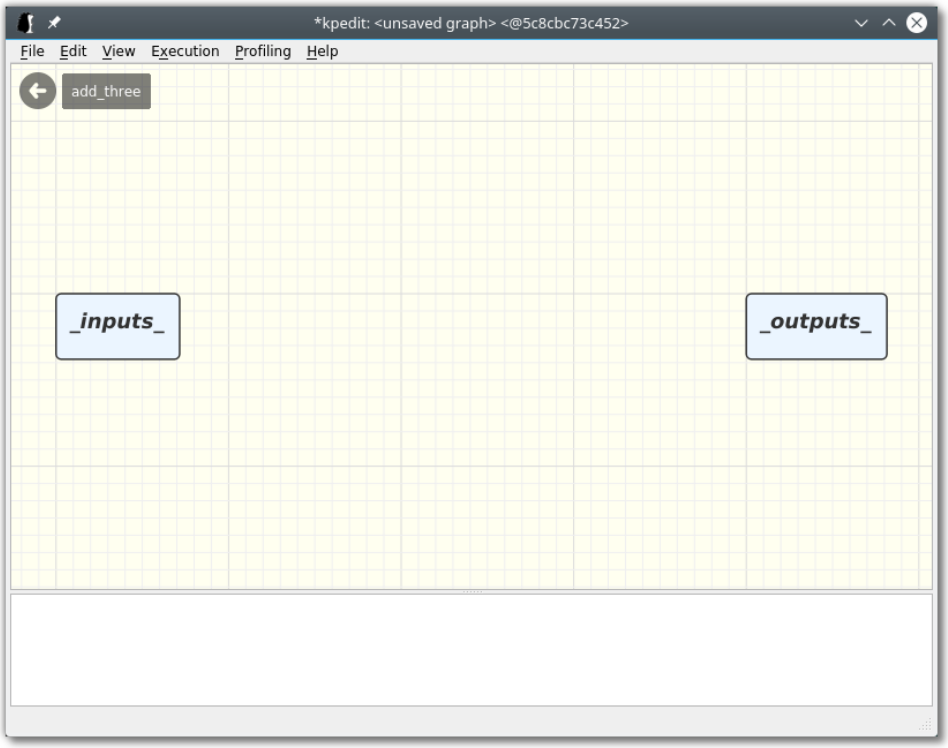
This tutorial will show you how create a subgraph. A subgraph is useful for packaging an arbitrarily complex process, defined by any number of graph nodes, into a single node, hiding the complexity of the process and making it easier to reuse it. Any corrections made later to the subgraph are then automatically propagated to the graphs that use it.

In this example, we will create a simple subgraph that adds three elements.

After creating a new graph, click with the right mouse button in the empty space in the middle of the graph and select *Create Empty Subgraph* from the popup menu.

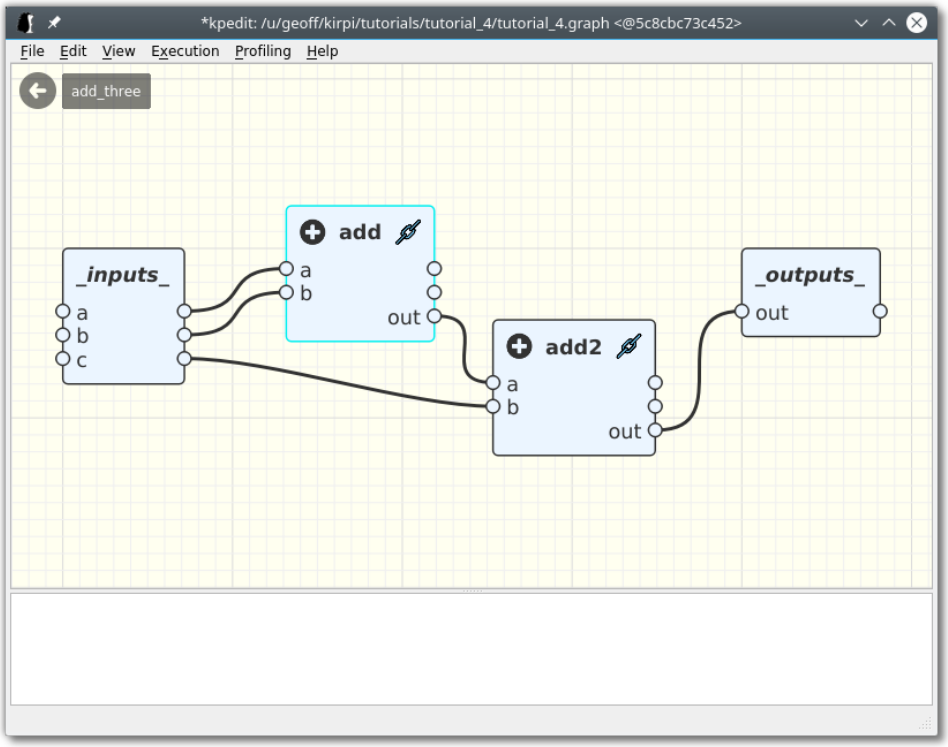


Rename the subgraph by right-clicking on the new subgraph node and selecting *Rename* from the menu and entering the name "add_three" in the dialog box. Then bring up the popup menu again and choose *Open Subgraph*.



The contents of the main graph are replaced by the contents of the subgraph. As you can see, the subgraph is in fact just another graph, with its own *_inputs_* and *_outputs_* nodes.

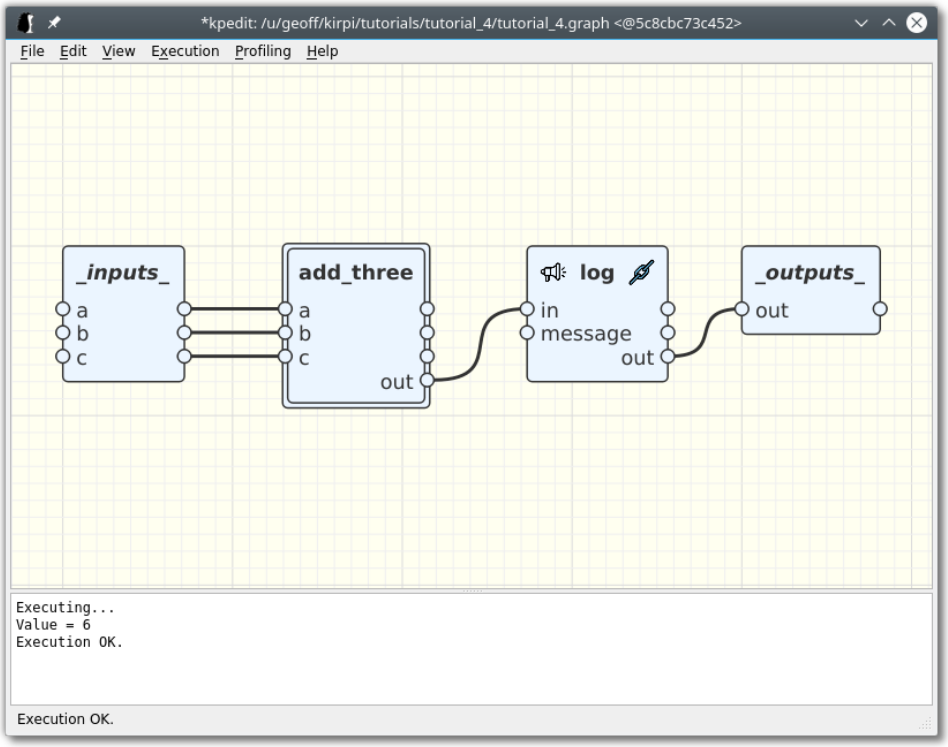
Add to the subgraph two *add* nodes, either by hitting the tab key and typing "add" in the box that appears, or by right-clicking in any empty space and choosing *Load Node* > *core* > *add* from the popup menu. Connect the nodes to obtain the following:



(Note that when you connect all the inputs from the *add* and *add2* nodes, the corresponding names in the *_inputs_* node will be *b* and *b2*, by default. You can rename *b2* to *c* by double-clicking on the *_inputs_* node, or by clicking on it with the right mouse button and selecting *Edit Node* from the popup menu.)

The subgraph is now complete. Exit the subgraph by right-clicking and choosing *Close Subgraph* from the popup menu, or by clicking on the arrow in the upper left corner of the graph, or by hitting the backspace key.

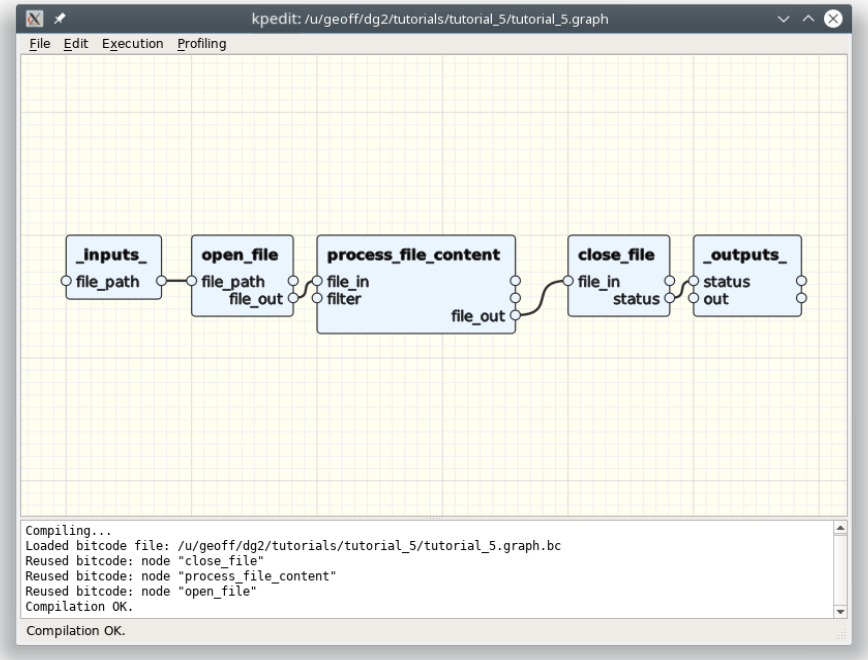
In the main graph, add a *log* node and connect it as shown. Set the graph's input values to 1, 2 and 3, then execute the graph to obtain the result shown below.



Tutorial 5: File Input/Output

In this example we will see how to read a text file and filter its contents.

Go to the `tutorial_5` directory and load the graph found there.



The parameter that is passed between the nodes *open_file*, *process_file_content* and *close_file* is a file. What is particular about this graph is that the object that is passed from node to node is modified by the nodes, whereas in the previous tutorials a new object was created by each node. The file object is initialized by *open_file*, modified by *process_file_content*, where data are read from the file, and closed by the *close_file* node.

Take a look at the execute function of the *open_file* node:

```
std::ifstream execute(const std::string& file_path)
{
    std::string prefix = std::getenv("KIRPI_ROOT_DIR");
    std::string path = prefix + file_path;
    kirpi::kpEmitInfo("opening file: " + path);
    return std::ifstream(path);
}
```

It returns a file input stream object (*std::ifstream*). Now look at the contents of the *process_file_content* node. It takes as input a reference to the file object, reads the contents of the file, which modifies the file object, and returns a reference to the modified object. No new object is created; the modifications are performed on the same object.

```
std::ifstream& execute(std::ifstream& file_in, const std::string& filter)
{
    std::string line;
    while (std::getline(file_in, line)) {
        if (line.find(filter) != std::string::npos) {
            kirpi::kpEmitInfo("Found: " + line);
        }
    }
    return file_in;
}
```

The node reads the entire file and displays lines containing the *filter* string.

Several points worth noting:

- In this example the value of the input *file_in* is modified as it traverses the graph. This behavior may be useful in cases where there are a lot of data to manipulate, to avoid duplicating the data.
- *file_out* is a simple copy of *file_in*. A node has to have at least one output, if it is to be executed, and the output has to be connected to another node or directly to the graph's *_outputs_* node. Because the node *process_file_content* does not actually return any values, we choose to return the input object, which can be reused by the next node.
- We use the function *kirpi::kpEmitInfo()*, which displays the contents of a string variable in kpedit's output window. The function is declared in the header file `kpglobal.h`, which is in the directory `KIRPI_ROOT_DIR/include/kirpi`.

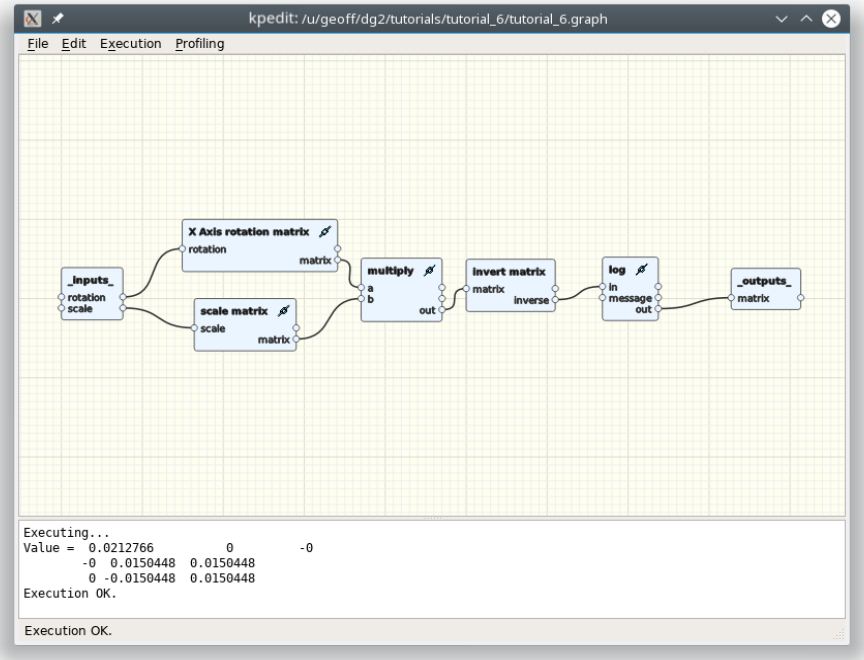
The *close_file* node should be self-explanatory.

If you execute the graph, it reads the contents of the `kpglobal.h` file and displays all the lines containing the string "void".

Tutorial 6: Using the Eigen Library

This tutorial will show you how to use the Eigen library in a Kirpi graph.

The graph multiplies a matrix representing a rotation about the X axis by a scale matrix and inverts the resulting matrix. The graph can be found in the `tutorial_6` directory.



Using directives:

If you edit the *X Axis rotation matrix* node, you will note that the *Using Directives* box contains the string:

```
using namespace Eigen;
```

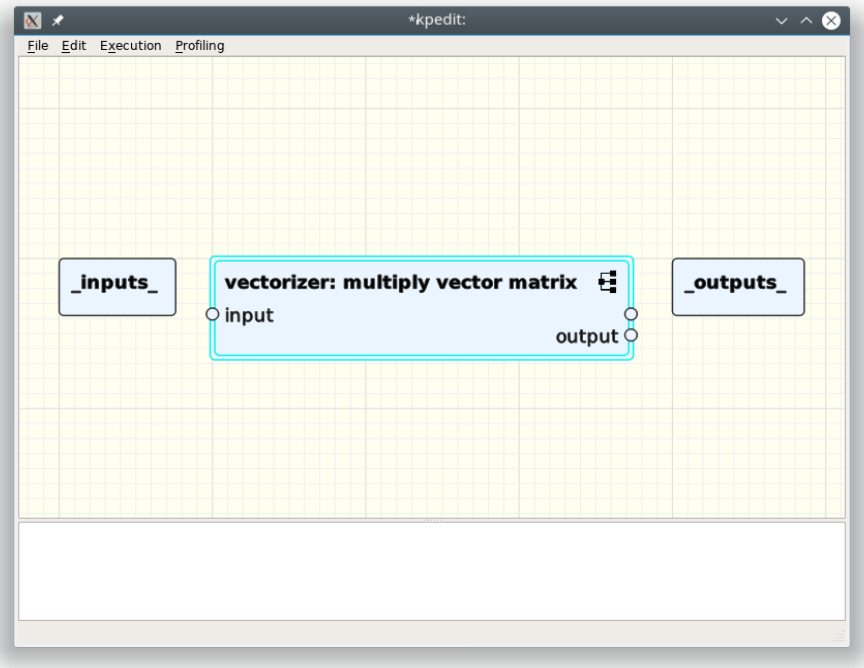
This makes it simpler to use the Eigen library in our example, since the *Eigen* namespace need not be prefixed to each identifier in the library.

Tutorial 7: Vectorizer Nodes

Vectorization consists of applying the same process to a collection of data to produce a new collection of data.

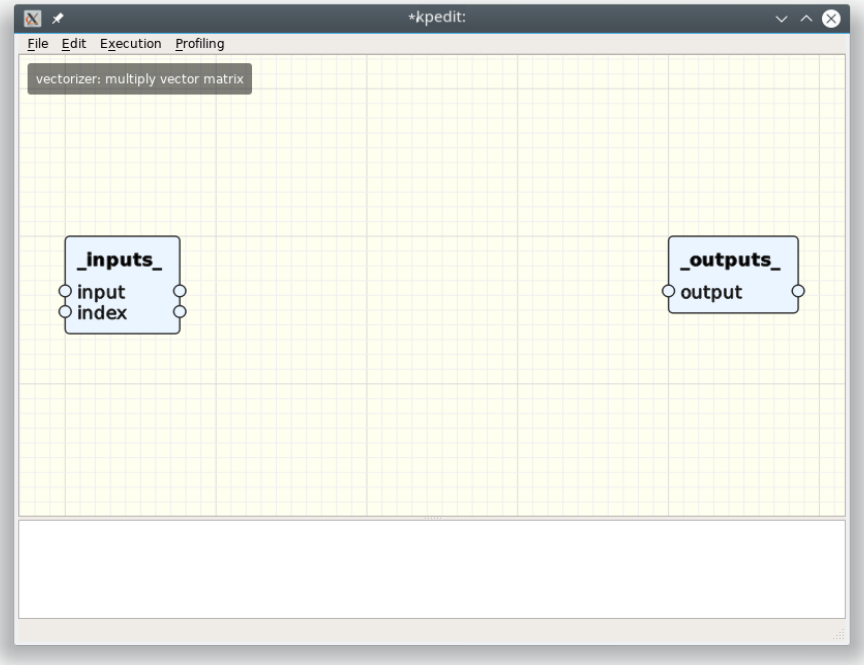
In this tutorial we will see how to multiply a list of 3D vectors by a matrix.

In *kpedit*, create a vectorizer node by right-clicking in the empty space in the graph and selecting *Create Vectorizer* from the popup menu. *kpedit* pops up a dialog box asking if you want the subgraph to have a "keep_result" output; just click on *No* for now. Name the node "vectorizer: multiply vector matrix".



The node has one input and one output. They are both of the same type: they are containers (*e.g.* vector, list, etc.). Note the little icon to the right of the node's name, indicating that it is a vectorizer node.

Vectorizer nodes are like subgraph nodes in that they contain a subgraph which you can edit by double-clicking on the node, or by right-clicking on it and selecting *Open Subgraph* from the popup menu.



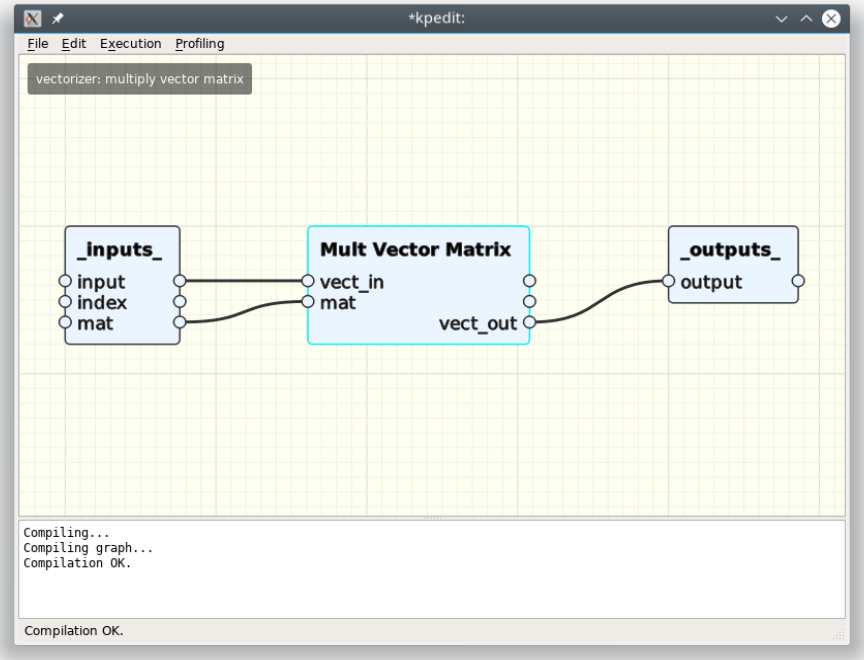
As you can see, the *_inputs_* node is different from those we have seen so far, in that it already contains two inputs: *input* and *index*. Whereas the input of the vectorizer node is a container, the input of the subgraph corresponds to a single element from that container, the index of which can be found in the input called "index".

Similarly, the *_outputs_* node contains an output corresponding to a single element which will be added to the vectorizer node's output container. (If you had clicked on *Yes* in the dialog box that appeared when you created the vectorizer node, there would be a second output called *keep_result*. That second output can be useful when the graph only generates results for some of its inputs: when *keep_result* is false, *output* is ignored.)

Create a node in the subgraph called "Mult Vector Matrix". The node will have two inputs (*vect_in* and *mat*) and one output (*vect_out*). The node's execute function will look like this:

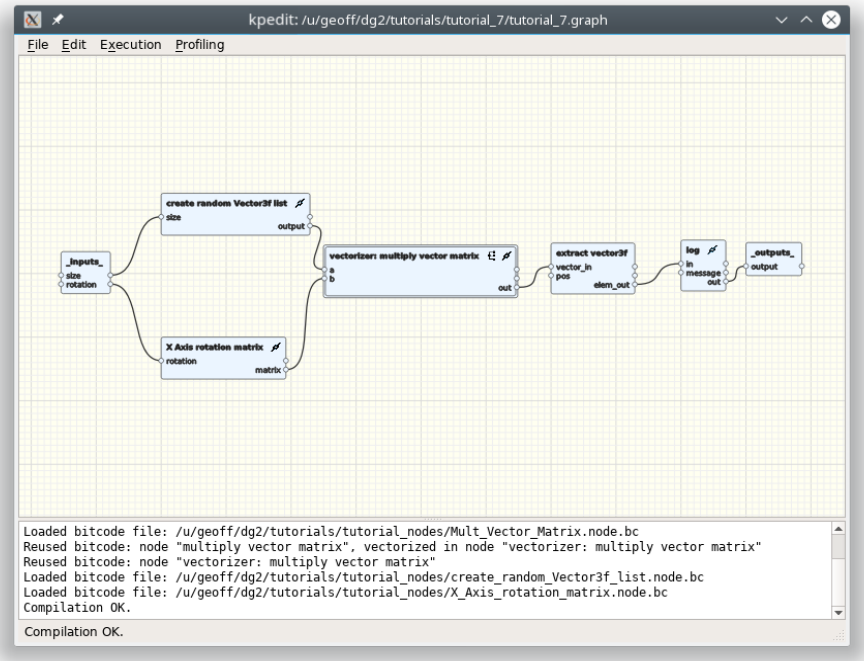
```
template<typename T1, typename T2>
void execute(const T1& vect_in, const T2& mat, T1& vect_out)
{
    vect_out = mat * vect_in;
}
```

Add an input to the subgraph's *_inputs_* node called "mat", then connect the nodes as shown below:



The vectorizer node is now complete. You need only connect a container of vectors and a matrix to its inputs, and each of the vectors in the container will be multiplied by the given matrix in multiple threads, using however many processors are available.

A complete example graph is provided in the directory `kirpi/tutorials/tutorial_7`.

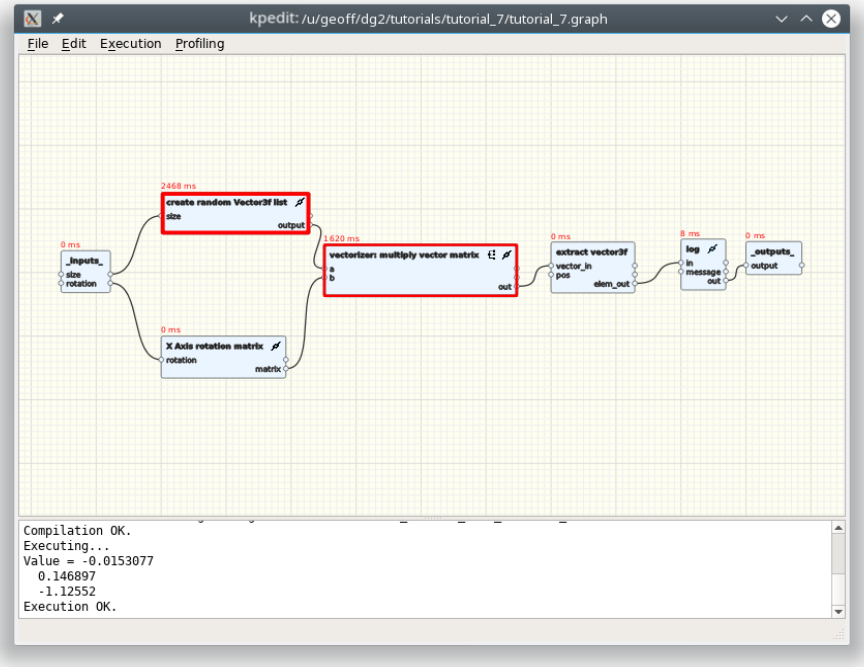


Tutorial 8: Profiling

In this tutorial we are going to reuse the graph from the preceding tutorial.

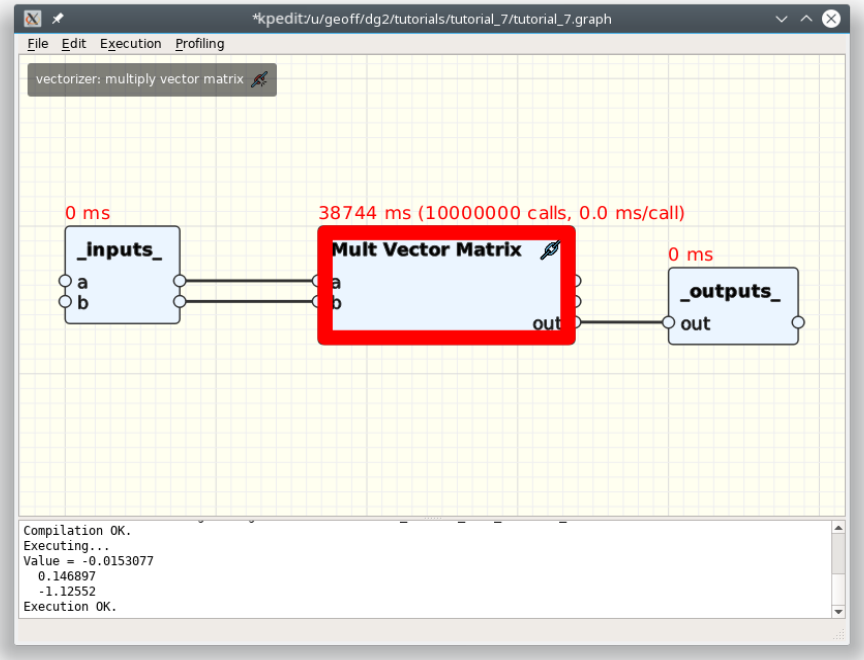
To turn on profiling, click on *Profiling* in the menu bar, then select *Enable Profiling*.

After executing the graph, you should see something like this:



Above each graph node, the time spent executing the node is displayed, in milliseconds. (If less than a millisecond is spent in a given node, 0 is displayed). Nodes with an execution time greater than a millisecond are highlighted with a red rectangle whose thickness is proportional to the node's execution time.

If the graph contains subgraphs, or vectorizer nodes containing subgraphs, you can go into a subgraph and see the profiling information for nodes in the subgraph, including the number of times each node was executed.



Note that the profiling option allows you to quickly detect which nodes in a graph or subgraph are the most expensive, but it does not profile a node's internal code.

Tutorial 9: Developer Workflow

We are aware that C++ developers generally have their own work environments and that they should be respected. Most developers use an IDE with an integrated version control system, and the design of a graph node for Kirpi should be possible within the same work environment.

The workflow that we suggest is as follows:

- The developer implements a node's C++ function in his usual work environment. The function can be tested there, without Kirpi.
- The *nodegen* utility makes it possible to automatically convert the C++ function into a Kirpi graph node. We will also see how to integrate *nodegen* into a build system based on CMake.

Let's take the case of a function that scales a matrix using the Eigen library. The `tutorial9.cpp` file contains the following code:

```
#include "Eigen/Dense"
#include "Eigen/Geometry"

using namespace Eigen;

void scaleMatrix(const float& scale, Matrix3f& matrix)
{
    matrix = Scaling(scale,scale,scale);
}
```

The *nodegen* utility can be used to generate a Kirpi node file directly from the source code:

```
nodegen -f scaleMatrix -I../../eigen tutorial_9.cpp scaleMatrix.node
```

The node file `scaleMatrix.node` can now be used in Kirpi graphs.

The directory `kirpi/tutorials/tutorial_9` also contains a `CMakeLists.txt` file which can be used to generate the node file as follows (from a Visual Studio command prompt on Windows):

```
cd tutorial_9
mkdir build
cd build
cmake ..
make
```

This generates the file `scaleMatrix.node`.

4 Using Kirpi

This section contains an overview of how Kirpi works, and how the *kpedit* application can be used to create, edit and execute graphs. For details about Kirpi libraries and their APIs, see the Doxygen-based HTML documentation found in the Kirpi installation's `doc` directory.

4.1 Graph Basics

Creating a Graph

A graph is a collection of nodes whose input and output plugs may be connected to one another. The graph itself, like each of its nodes, may have any number of inputs and outputs. Once a graph has been compiled, the user can set the graph's input values, execute it and retrieve its output values without recompiling it.

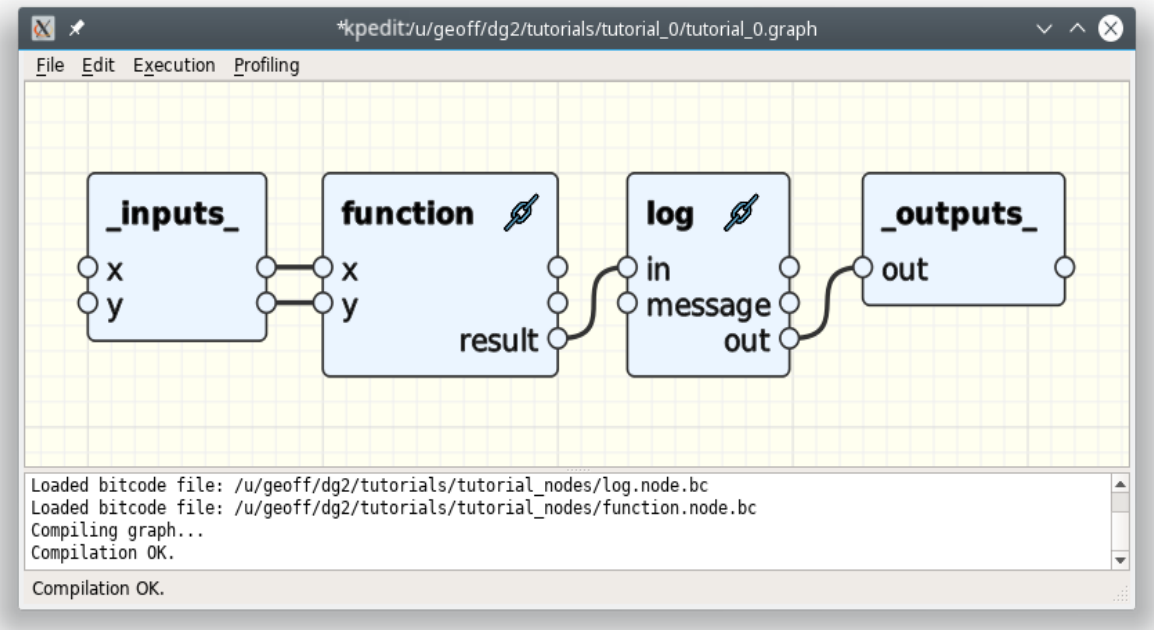


Figure 1: Display of a graph in kpedit

The simplest way to create a graph is to use the *kpedit* application, which provides a graphical user interface for editing graphs. You can also use the libkirpi API to create graphs, or write C++ graph functions in an ordinary text editor.

A graph (or subgraph) is always composed of at least two nodes: an _*inputs*_ node and an _*outputs*_ node.

Creating a Graph Node

A graph node is composed essentially of connectable inputs and outputs (plugs), an execute function that uses the inputs to compute the outputs, and parameters for compiling the function.

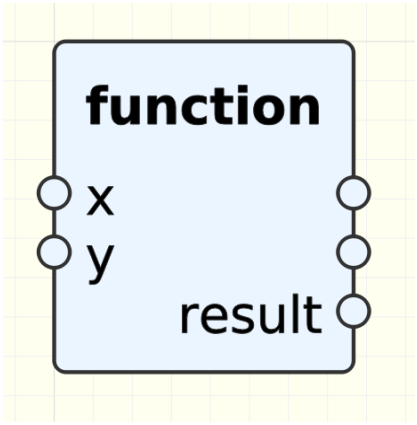


Figure 2: Display of a node and its input/output plugs in kpedit

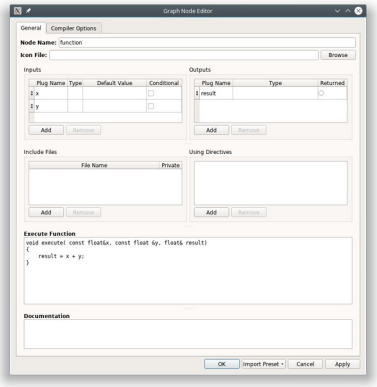


Figure 3: Display of a node's parameters and execute function in kpedit

A node can be saved to a file and shared among several graphs and users, or it can be defined directly in the graph: see Instance Nodes below.

A node's execute function is defined by C++ code stored in the node. The C++ code must define a function called "execute", although it may define other functions or classes as well. As a rule, the code cannot include header files, however. If it needs header files, they must be added in the *Include Files* box in the node editor (figure 3). The execute function can make calls to other libraries if the necessary header files are included, and if any other required compilation parameters have been set up in the node editor.

The execute function may or may not return a value for one output plug. If it does return a value, the associated output plug must be marked *Returned* (see Tutorial 3: Returning a Value).

In the execute function, the names of the parameters must be **exactly the same** as those declared for the input/output plugs. If you change the name of a plug, you must change the name of the corresponding parameter in the execute function.

For more information about writing the execute function, see Tutorial 1: Adding Character Strings and the other tutorials.

Input and Output Nodes

Every graph (and subgraph) must contain two nodes defining the graph's inputs and outputs, called *_inputs_* and *_outputs_*. Unlike normal graph nodes, the input and output nodes have no execute function.

If nothing is connected to the graph's *_outputs_* node, the graph has nothing to do and will not execute.

You can have a graph with no inputs, however.

Input and Output Plugs

A graph node can have any number of input and output plugs; when the node is executed, it takes the values of its inputs and computes the values of its outputs. The node is executed only if it is connected to the graph's output node (directly or indirectly).

An input plug can be connected to an output plug from another node in the graph, to retrieve a value computed by that node. In that case, when the graph is executed, the upstream node will be executed first, so that the values of its outputs are available for use by the downstream node. An input plug can also be connected to an input plug from another node. In that case, however, the input takes its value from the other input plug, but the other node is not executed (see figures 4 and 5).

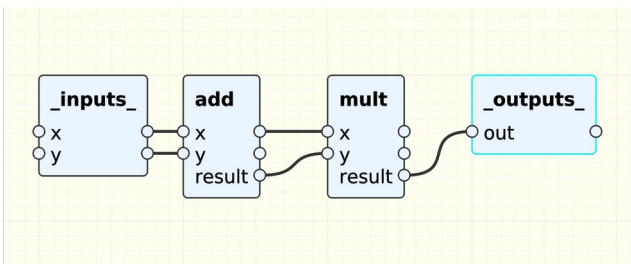


Figure 4: add *and* mult *are executed*

The *mult* node uses the result of the *add* node and the graph's *x* input.

When the graph is executed, the *add* node is executed first, followed by the *mult* node.

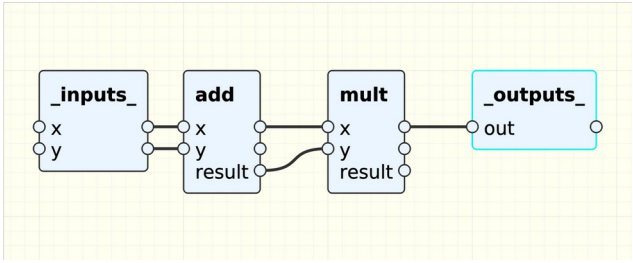


Figure 5: add *and* mult *are ignored*

The *mult* node uses the output of the *add* node and the graph's *x* parameter, but its output is not connected to anything.

When the graph is executed, neither the *add* node nor the *mult* node is executed: the *x* input is copied directly to the *out* output.

Connecting an input plug to another input plug allows you to retrieve the same input value, but does not trigger the execution of the node. A node is executed only if one or more of its outputs are connected directly or indirectly to the graph's *_outputs_* node.

For more information, please read the chapter **Input and Output Plugs** in the libkirpi documentation.

Compiling a Graph

In *kpedit* a graph is compiled:

- when the user selects *Execution → Compile* from the menu bar;
- each time the graph is modified, if *Auto Compile* is turned on (see *Execution → Auto Compile* in the menu bar);
- before the graph is executed, if necessary.

Kirpi uses LLVM and Clang to compile a graph in memory. That is, for each node with an execute function, and for the graph itself, Kirpi generates code which it then compiles using a JIT (Just In Time) compiler. The C++ code is first compiled into LLVM's *intermediate representation (IR)*, also known as *bitcode*, which is saved and reused, and then converted into machine code optimized for the machine that will be executing the graph.

Generating an intermediate representation of the code has the following advantages:

- It allows optimizations which would not be possible otherwise.
- The machine code generated is optimized for each machine that executes the graph so that it can fully exploit each machine's capabilities.

For more information, see the chapter **Compiling a Graph** in the libkirpi documentation.

Executing a Graph

Once a graph has been compiled in memory, the execute function it defines can be called within the same process, just like any other function that is part of the running program or of a library that it is

linked to. Its input values can be modified, it can be executed again, and its output values retrieved any number of times. And like any other function it can potentially be traced with a debugger or profiled (sort of).

4.2 Graph Nodes in Detail

In the previous chapter on Graph Basics, we discussed simple graph nodes (a node with a single execute function that reads its inputs and computes its outputs), but graph nodes can do much more. Subgraph nodes contain not just an execute function but another graph, composed of any number of nodes. Vectorizer nodes can be used to adapt an existing node (or subgraph) so that it can process multiple inputs in parallel, in multiple threads. And using conditional inputs, nodes can be designed to choose which of their inputs are needed, and avoid computing the other inputs.

In this chapter we explain how subgraph nodes, vectorizer nodes and conditional inputs work. We also explain the mechanism whereby nodes are saved to files that can be shared by any number of graphs (instance nodes), and discuss compiler options and other information that a graph node may need for compilation.

Subgraph Nodes

Subgraphs make it possible to simplify a graph by breaking it up into smaller modules that are easier to manage.

Like a top-level graph, a subgraph always contains an *_inputs_* node and an *_outputs_* node. To create a subgraph in *kpedit*, just select *Create Empty Subgraph* from the popup menu.

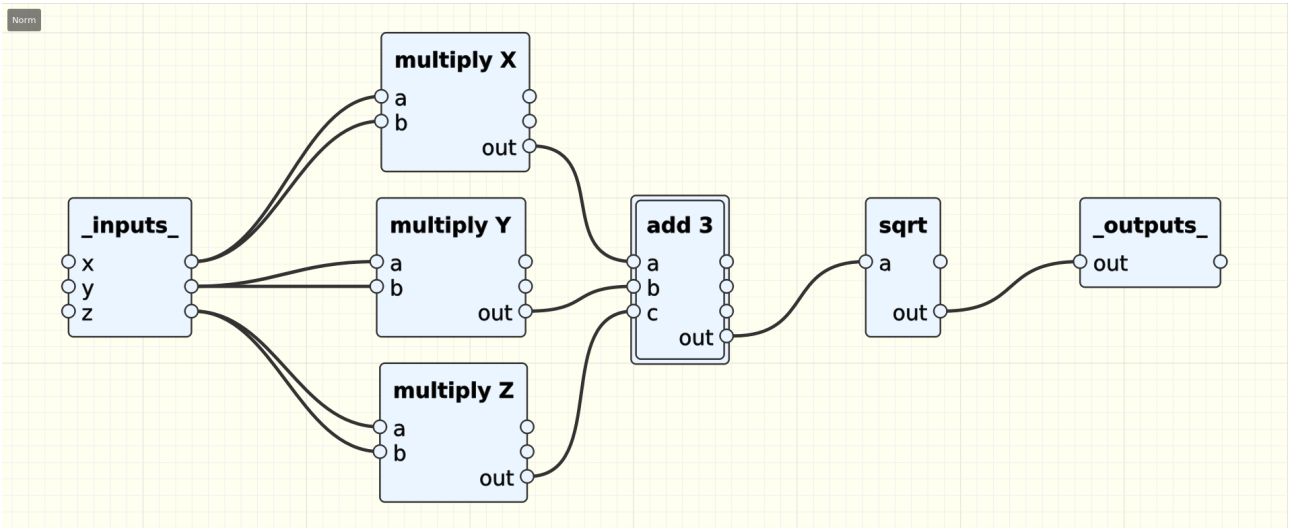


Figure 6: Editing a subgraph in kpedit. *The name of the subgraph node ("Norm") appears in the upper left corner.*

Subgraphs may themselves contain other subgraphs. The name of the subgraph node being edited is displayed in the upper left corner of the window, along with the name of the subgraph node containing it, and so on. Inputs and outputs added to the subgraph are automatically added to the subgraph node containing it:

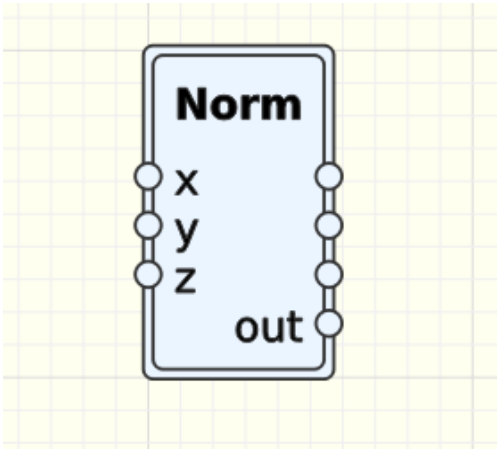


Figure 7: A subgraph node.

For a more practical explanation of subgraphs, see Tutorial 4: Subgraphs.

Vectorizer Nodes

It is sometimes useful to be able to apply a subgraph to a collection of data elements to produce another collection of elements. For instance, to deform a 3D surface, you might want to start with a subgraph that deforms a single point. Once that subgraph is done, you would like to be able to apply it automatically to all the points on the surface: feed it a vector of points, and get a new vector of transformed points in return.

This is what Kirpi vectorizer nodes do. This approach makes it possible to separate the algorithm (the subgraph) from the data to be processed. The vectorizer node can also deal automatically with multithreading, doing the work on multiple processors when they are available.

An existing subgraph can be converted to a vectorizer node, or the subgraph can be created from scratch.

We recommend that you start with the example provided in Tutorial 7: Vectorizer Nodes. For more information about supported container classes and how to extend vectorizer nodes to other classes, see the **Vectorizer Nodes** chapter in the libkirpi documentation.

Instance Nodes

Each of a graph's nodes can be defined in one of two ways: the node can be defined directly in the graph file, or the graph can contain a reference to a separate node file. In the latter case, the node is called an *instance* node, because the graph may contain several instances of the same node, as can other graphs. If the node file is modified, all instances will be modified, too.

In *kpedit*, when you select *Create Node*, the resulting node is defined directly in the graph file. To create an instance node, there are two ways to go about it:

- Select *Load Node* or *Browse Nodes* from the popup menu and choose a node file.

- Or, press the tab key and, in the widget that pops up, start typing the name of a node file. The widget knows about all the node files present in directories identified by the NODE_PATH configuration variable (see Configuration).

Instance nodes in *kpedit* are identified by a little link icon which appears to the right of the node's name. Also, if you position the mouse over a node and wait a second or two, a *tool tip* appears with more detailed information about the node, including the path of the file that the node is an instance of.

Note that, in the course of editing a graph in *kpedit*, instance nodes may become uninstanced and vice versa:

- If you modify the contents of an instance node, the node is "uninstanced" so that other instances of the instanced graph file will not be affected. When that happens, the link icon next to the node's name is replaced by a broken link. And the node's tool tip is modified to show the full path of the file that the node used to be an instance of. You can see how the node has been changed by selecting *View Differences* in the popup menu, and you can restore the original node by selecting *Revert Changes*.
- If you save a non-instance node to a file (by selecting *Write Node to File* in the popup menu), the node becomes an instance of that file. Any other nodes that reference that file, in the same graph or in other graphs, are updated, too. So if you edit an instance node and would like the change to affect other instances, remember to write the node back to the original file when you finish.

Note that non-instance nodes may make compilation of the graph slower than instance nodes.

Conditional Inputs

Normally a node's inputs are evaluated before the node is executed, and the resulting values are passed to its execute function. In some cases, though, this is not desirable; the node may want to decide for itself which inputs it needs to evaluate. Imagine an "if/else" or a "switch" node, for example, that copies just one of its inputs to an output plug, depending on some condition. It would be a waste of time for it to evaluate the unused inputs.

To handle this scenario, an input may be declared conditional, by checking its *Conditional* check box. When an input is conditional, the execute function is passed an instance of std::function rather than a value. The function takes no parameters and returns a constant reference to the type of the input. So a node with two conditional inputs of type `float`, for example, might look like this:

```
float execute(
    bool which, std::function<const float&(void)> in1,
```

```
    std::function<const float&(void)> in2)
{
    return which ? in1() : in2();
}
```

For more examples, see the predefined *if* and *switch* nodes in the Kirpi installation's `nodes` directory.

Include Files, Compiler Options and Using Directives

Like any C++ compiler, Kirpi requires a certain number of parameters specifying, for example, where header files and libraries are located, and defining other compiler options.

These options for each graph node can be set in *kpedit* as illustrated in Tutorial 6. Options that apply to the entire graph can be set by selecting *Edit → Edit Graph* in the menu bar. More detailed information is available in the libkirpi documentation: see the section **Compiling a Graph**.

Note that when you create a graph node, you can import presets containing predefined compiler options for a given situation.

We recommend that you define presets to simplify the creation of graph nodes that use a given library or application.

Customizing a Node's User Interface

Kirpi's graph editor (used in *kpedit*, but available for use in any Kirpi plug-in or application) allows the user to edit the default values of a node's inputs, that is, the values that are used when the inputs have no upstream connections. Normally the user interface for editing those values is rudimentary: a simple text entry widget for each input. But if you wish to provide a custom-designed user interface for a given node, you can use the Qt Designer application to do so.

The mechanism works as follows. In Qt Designer, create a *form* to be instanced and displayed when the user edits the node's values, containing a Qt widget for each of the node's inputs. Save the form to a UI file in the same directory as the `.node` file, and with the same name, plus the `.ui` extension. (This mechanism only works for instance nodes defined in their own `.node` files.) That is, if the graph node is defined in a file called `sgnork.node`, its UI file must be called `sgnork.node.ui`.

The UI file must contain a widget for each of the node's input plugs. The widget for a given input is identified by its object name, which must be the same as the name of the input plug. If the widget is a subclass of one of the widget types supported by Kirpi, then it is used to display and edit the value of the corresponding input. Supported widget types are:

- QLineEdit
- QComboBox
- QAbstractButton (*e.g.* QCheckBox, QRadioButton)
- QAbstractSlider (*e.g.* QDial, QSlider)
- QSpinBox
- QTextEdit

- QPlainTextEdit

The UI file can contain other widgets as well, of any class, but those widgets cannot be associated directly with an input plug.

For inputs of type string, whose default values are usually surrounded by quote marks, you may want to hide the quote marks when their values are displayed in a widget (*e.g.* a QLineEdit or a QComboBox). If you create a boolean property called "hideQuotes" for the input's widget and set it to true, quote marks will be stripped from the value when it is displayed, and quote marks will be added to values supplied by the user.

Note that it is the user's responsibility to update a node's UI file if input plugs are added to or removed from the node, for example, or if inputs are renamed. Kirpi's graph editor will never modify a UI file.

4.3 Sticky Notes and Backdrops

Sticky notes and backdrops are helpful for documenting and organizing complex graphs. A sticky note is simply a note that can be placed anywhere in the graph. A backdrop can be used to group together some number of related graph nodes, setting them apart visually and making it possible to manipulate them together. Sticky notes and backdrops have no effect on how a graph actually works.

Sticky Notes

Sticky notes are just notes that can be placed anywhere in the graph. They might be used to explain how the graph works, for example, or to identify changes that ought to be made in the future: see Figure 8: Sticky notes.

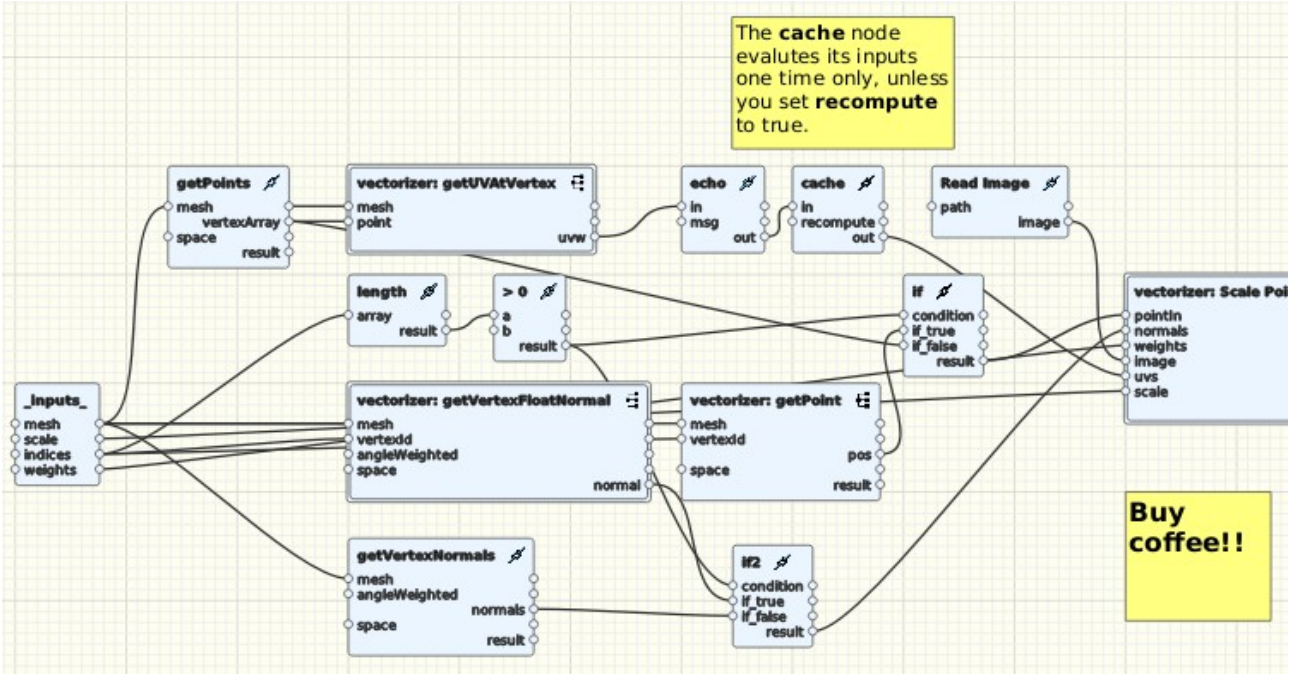


Figure 8: Sticky notes.

To create a sticky note in *kpedit*, right-click in some empty space in the graph, then select *Add Sticky Note...* from the popup menu. The dialog box that appears allows you to choose the sticky note's color and font, and to enter the text it will contain. Note that the text can contain HTML tags, such as `<b>` for bold text and `<i>` for italics.

You can resize a sticky note by clicking on one of its edges or corners and dragging it; you can move it by clicking and dragging anywhere inside the sticky note. To edit the text or to change the color or font, right-click on the sticky note and select *Edit Sticky Note...* from the popup menu.

Backdrops

Backdrops are similar to sticky notes, in that they are colored boxes containing text and have no effect on what the graph does. But backdrops can be used to organize complex graphs into distinct zones: see Figure 9: Backdrops.

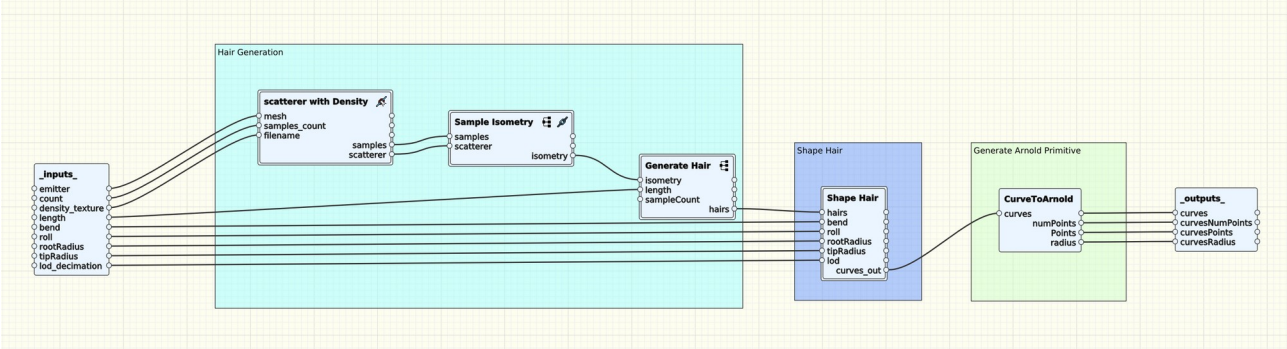


Figure 9: Backdrops.

Backdrops always appear behind other graph elements (*e.g.* graph nodes and sticky notes). When you move a backdrop, any elements contained in the backdrop move with it.

Note that to move a backdrop (and everything it contains), you cannot just click anywhere inside it: you must click on the **top** of the backdrop. This is so that you can easily select nodes in front of a backdrop by clicking and dragging inside the backdrop.

There are a few different ways to create backdrops in *kpedit*. One is to right-click in some empty space, then select *Add Backdrop...* from the popup menu. A dialog box similar to the one for sticky notes appears, allowing you to choose the backdrop's color and font, and to enter the text it will contain. You can then resize the backdrop, so that it contains the graph elements you want to keep together, by clicking and dragging on its edges or corners.

A second way to create a backdrop is to select one or more existing elements, right-click and then select *Add Backdrop Behind Selection...* from the popup menu. The new backdrop will be positioned and sized automatically so that it contains whatever was selected.

A backdrop can contain other backdrops as well. That is, if a big backdrop contains a large set of nodes, you might want to use smaller backdrops to group together subsets of those nodes: see Figure 10: Layered backdrops.

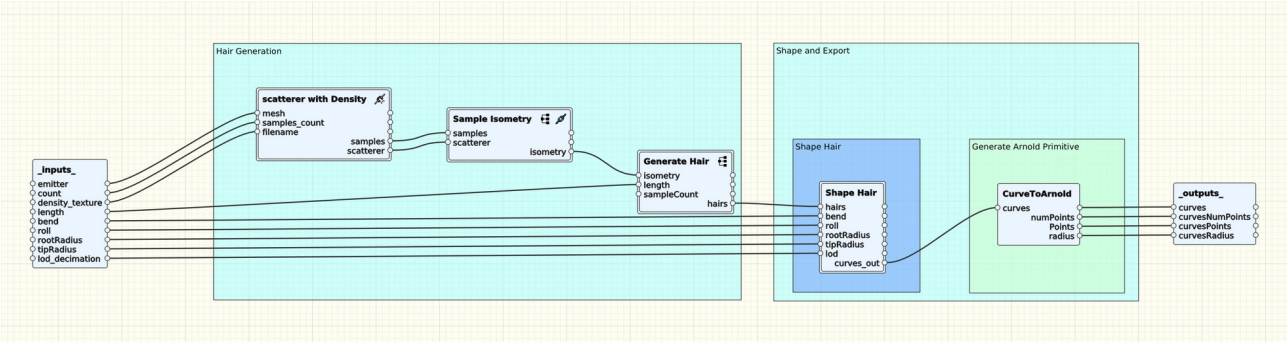


Figure 10: Layered backdrops.

If you select a subset of nodes and then click on *Add Backdrop Behind Selection...* in the popup menu, *kpedit* will automatically create a smaller backdrop on top of the existing backdrop.

Each backdrop has a *depth* value that determines whether it appears in front of or behind other backdrops. Backdrops with higher depth values appear in front of backdrops with lower depth values. By default, backdrops have a depth of zero; to edit a backdrop's depth, right-click on it and select *Edit Backdrop...* from the popup menu. The dialog box shows the backdrop's depth as well as its text, color and font.

4.4 Graph Controller Window

The Graph Controller Window is a handy graphical user interface (GUI) for testing a graph: setting its inputs, executing the graph and viewing its outputs.

The *kpexec* program provided with Kirpi allows you to set a graph's inputs and retrieve its outputs, too. But its text-based user interface is rudimentary and can be cumbersome to use. *kpedit*'s main window also allows you to execute a graph, but it is not great for viewing the values of its inputs and outputs.

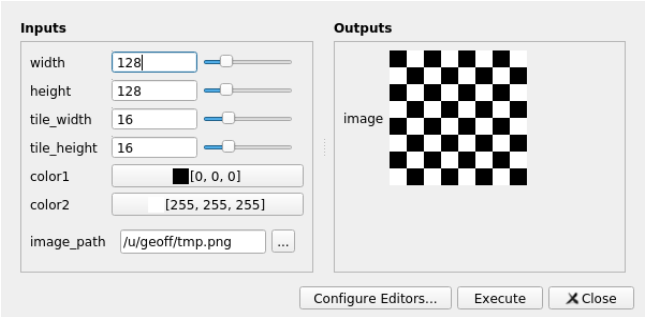


Figure 11: Graph Controller Window example.

The Graph Controller Window allows you to:

- set the current values of the graph's inputs without modifying their default values
- execute the graph automatically when an input changes, without recompiling
- visualize the values of the graph's outputs
- provide plug-ins for editing and viewing input and output values of any type

The Graph Controller Window is available in the *kpedit* application (*Execution → Show Controller Window* in the menu bar), or via the `GraphController` class for your own applications and plug-ins.

Using the Graph Controller Window

The Graph Controller Window contains an editor widget for each of the current graph's input and output plugs: inputs on the left, outputs on the right. Whenever you change the value of an input, the graph is executed automatically and the values of the outputs are updated. You cannot edit the output values (their widgets are read-only).

It is important to note the difference between an input's *current* value and its *default* value. When you edit the graph's *_inputs_* node, you modify the types and default values of the input plugs, causing the graph to be recompiled. When you edit input values in the Graph Controller Window, you only modify the input plugs' current values in memory. This difference has several implications that may not seem obvious at first glance:

- You cannot edit input values in the Graph Controller Window until the graph has been compiled.
- Editing input values in the Graph Controller Window does not cause the graph to be recompiled.
- If the graph is recompiled (*e.g.* because it has been modified), then the values of its inputs revert to their default values.

The editor widgets provided by default perform simple text-based entry and display. If a plug is a scalar value of a built-in type (*e.g. int*, *float*, *char*, etc.) or an *std::string*, its widget displays the plug's value and, for an input plug, allows you to edit it. For all other plug types, the widget just displays the type of the plug; you cannot edit it.

Where the Graph Controller Window becomes more interesting, though, is that it allows you to provide different editors better adapted to different plug types. Editors for input plugs might use combo boxes or sliders, for example; editors for output plugs might display images or even 3D scenes. A number of examples are provided with Kirpi; a plug-in mechanism allows you to develop your own.

Other Editor Widgets Provided by Kirpi

Kirpi provides a number of alternative editor widgets for the Graph Controller Window. They are described briefly here; the source code for each can be found in the `examples/plugins` directory in the Kirpi installation.

slider. A simple slider that can be used to enter numeric values.

spin_slider. A slider combined with a text widget, allowing the user to enter numeric values either with the slider or by typing them.

check_box. A check box that can be used to toggle a boolean value.

combo_box. A combo box that can be used to present the user with a number of predefined choices.

file_picker. A widget allowing the user to select the path of a file or directory. The user can either type or paste the path, or click on a button to bring up an interactive file browser.

color_button. A push button which displays a color, both as a colored swatch and as RGB or RGBA components. Clicking on the button brings up a color picker.

image. Displays a 2D image in memory (more specifically, the OpenImageIO type *ImageBuf*).

Some editor widgets accept one or more parameters to adapt their behavior for a given plug: the entries of a combo box, for example, or the minimum and maximum values of a slider. Those parameters can be set for each plug by clicking on the *Configure Editors…* button.

Configuring Editors

If you click on the *Configure Editors…* button at the bottom of the Graph Controller Window, a dialog box is displayed. The dialog box allows you to specify, for each of the graph's input/output plugs, which plug-in editor widget should be used for it (if any), and any parameters to be passed to it.

The dialog box looks something like this:

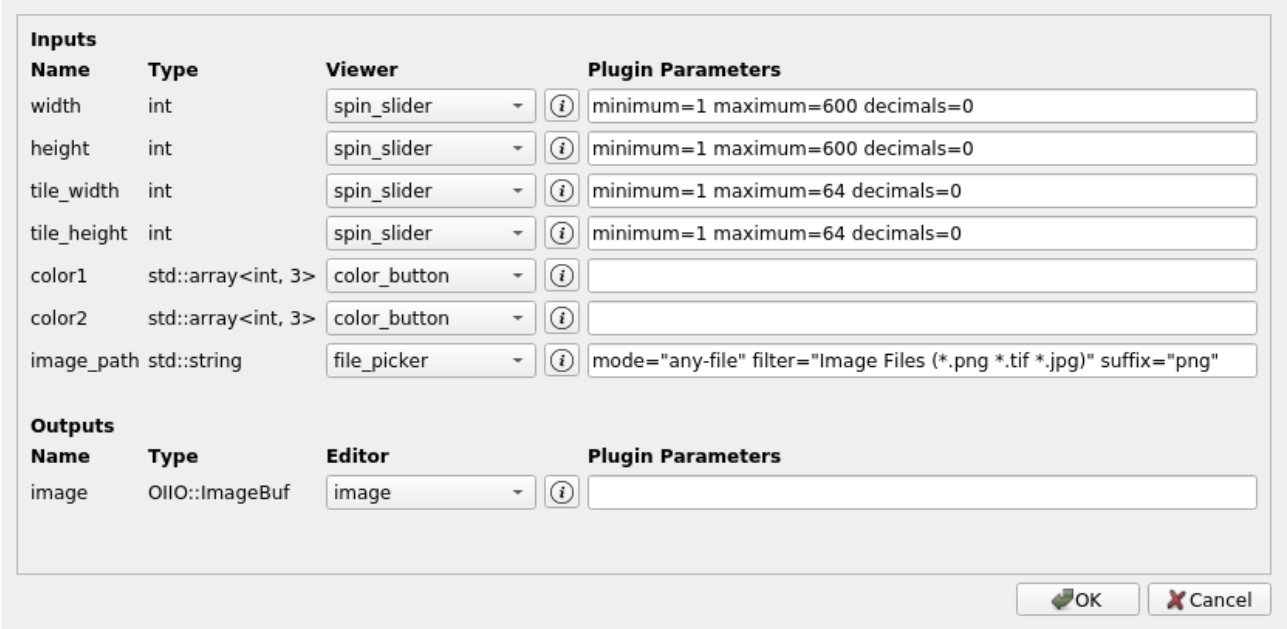


Figure 12: The Configure Editors dialog box.

The dialog box contains a line for each input/output plug, showing its name and its C++ type. The combo box to the right of the plug's type allows you to choose from the available editors, or *[default]* to use the default text-based editor. Clicking on the *i* button displays any documentation provided by the editor selected in the combo box, which generally includes which C++ types the editor supports, and information about any parameters that might be passed to it. Those parameters, if any, can be entered in the text box to the right.

Creating Your Own Editors

As mentioned above, the source code for each of the editor widgets provided by Kirpi can be found in the `examples/plugins` directory. Users are encouraged to develop their own editor plugins, using that source code as a model.

A plugin must implement the interface defined by the class `ValueEditorPlugin`: see the header file `kpui/plugin.h`. The examples/plugins directory also contains a `CMakeLists.txt` file showing how to compile a plugin with CMake. The compiled plugin (a DSO on Linux, a DLL on Windows) must then be placed in a directory where Kirpi searches for plugins; the PLUGIN_PATH variable in the Kirpi configuration file can be used to designate any number of plugin directories.

4.5 Advanced Graph Topics

Parallel Execution

Parallelization within a Kirpi graph is implemented automatically by vectorizer nodes, but it may also be useful to parallelize the execution of an independent branch of the graph. To that end, Kirpi provides a special type of node called a *parallel* node.

A *parallel* nodes executes two or more inputs in parallel. That is, it creates a thread for each input and executes those threads concurrently. When all the threads have finished, the node copies its inputs to its outputs (a parallel node always has exactly one output per input).

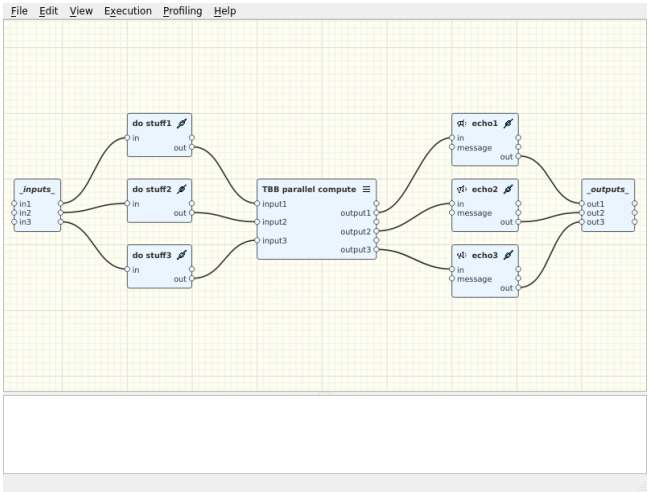


Figure 13: Multithreading with a parallel node.

To create a parallel node in *kpedit*, click with the right mouse button in an empty space and select *Create Parallel Node* from the popup menu. You cannot edit a parallel node like a normal graph node; to add an input to it, you must click and drag the output of another node, and release the mouse button over the parallel node. Each time you do this, an input/output pair is added. To remove inputs/outputs that are no longer needed, click on the parallel node with the right mouse button and select *Remove Unused Inputs* from the popup menu.

N.B. It is important to note that multithreading in a graph requires not only the spawning of multiple threads, but also that the code executed in those threads be thread safe and scalable. A thorough understanding of the principles behind multithreading is necessary to fully exploit it.

Parallel nodes can use either Threading Building Blocks (TBB) or OpenMP to spawn and manage concurrent threads. By default they use TBB, but in *kpedit* you can change that by right-clicking on the node and selecting *Multithreading Mode → OpenMP* from the popup menu. Note however that, for now, Kirpi cannot perform profiling on graphs containing OpenMP-based parallel nodes. When one or more OpenMP-based parallel nodes are used in the graph, profiling is disabled automatically.

You can also disable multithreading altogether in a parallel node, *e.g.* for debugging purposes, by selecting *Multithreading Mode → [Off]* from the same popup menu. In that case the parallel node will evaluate each of its inputs in turn, in a single thread.

Graph Node Caching

When a graph may potentially be executed many times, it may be useful to cache some of the results it computes. In any computation that requires a long initialization phase, in particular, where the initialization must be performed once but can then be reused any number of times, caching can be beneficial.

To facilitate caching of results, Kirpi provides a template-based *cache* node that can be used to cache any type of data. For more information about the *cache* node, see the **Special Inputs** chapter in the libkirpi documentation.

Debugging

For information about debugging graphs, please see the **Debugging** chapter in the libkirpi documentation.

Profiling

The *kpedit* application provides an integrated profiling mechanism which can be activated by selecting *Profiling → Enable Profiling* from the menu bar.

When the graph is executed, the time spent in each node is then displayed above the nodes, and a red box whose thickness is proportional to that time is drawn around each node (the more time spent in the node, the thicker the box's edges).

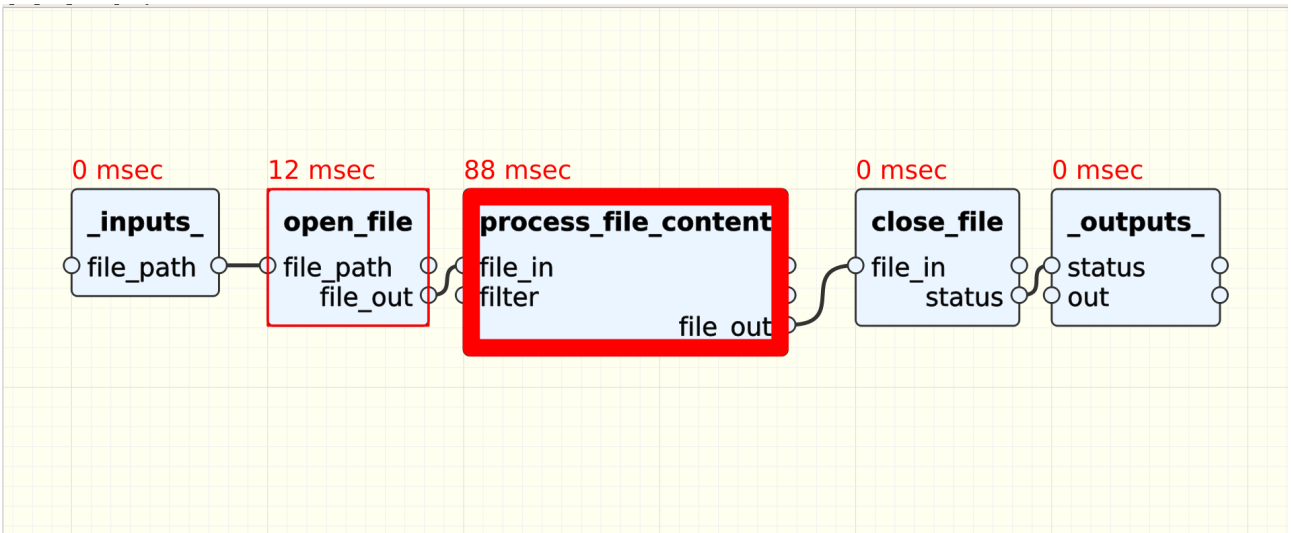


Figure 14: Profiling in kpedit

Profiling information is also available in subgraphs and in vectorizer nodes. For more information, please see the **Profiling** chapter in the libkirpi documentation.

Black Box Graphs

If you have spent a lot of time and effort developing a Kirpi graph, you might want to share it with other users, but you might not want them to be able to copy it or make changes to it. Black box graphs are Kirpi's solution to that problem.

When a graph is exported as a black box, all nodes other than the graph's input and output nodes are removed. Because the bitcode generated for the graph is preserved, the black box continues to work exactly like the original graph. Unlike the original graph, however, it cannot be edited.

In *kpedit* a black box graph looks something like this:

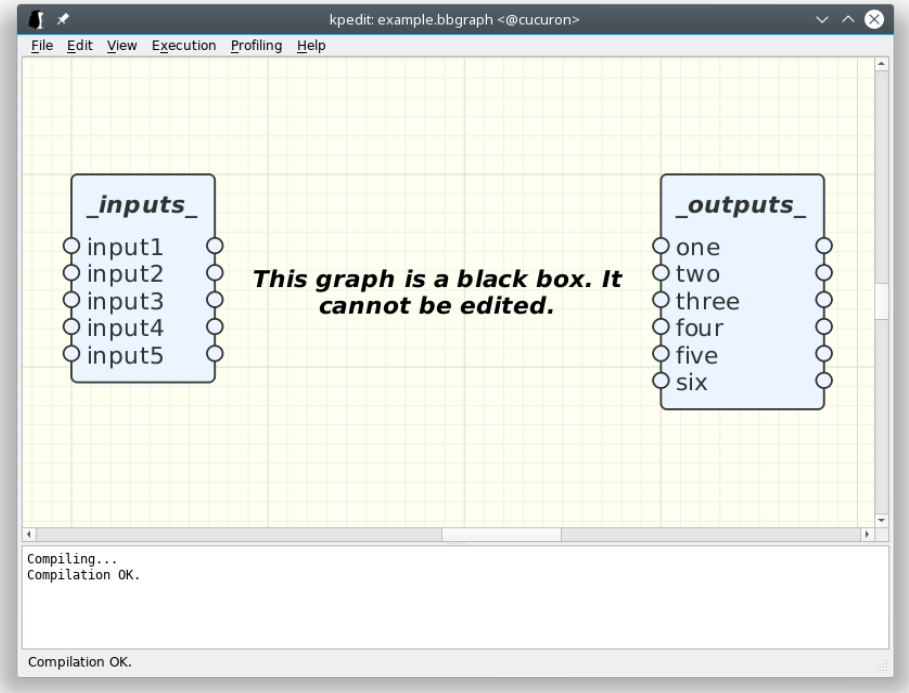


Figure 15: A graph exported as a black box

All the user can do with a black box graph is set the values of its inputs, execute it and retrieve the outputs.

To export a graph as a black box from *kpedit*, select *File→Export as Black Box...* from the menu bar, then enter the path of the black box graph file to create. You can also export a black box from *kpexec* via the `blackbox` command, or from the API via the `Graph::exportBlackBox()` method.

Whatever method you use, note that Kirpi actually creates two files: the black box graph itself, in a file with the extension `.bbgraph`, and the graph's bitcode, in a file with the same name plus a `.gbc` extension (graph bitcode), in a subdirectory whose name begins with `bitcode-linux` or `bitcode-windows`. Be sure to keep both those files; one will not work without the other.

Note also that there is no way to recover the original graph from a black box, so be sure to keep the original, too.

4.6 Development Methodology

Several different strategies can be imagined for developing Kirpi graphs. The choice depends on the complexity of the graphs and the work habits of the developers.

It is our belief that the nodes of a graph are rarely a one-to-one transcription of the functions of a C++ API. Graph users need functionality that is easy to use, not lots of low-level nodes that must be combined in order to do something useful with them.

Approach 1: Developing in *kpedit*

Nodes are developed directly in *kpedit*'s graphical user interface. When they are ready, they are saved to node files that can be used by other users or in other projects. The saved files can be version controlled using an appropriate tool, such as Git, Perforce or Subversion.

Approach 2: Developing in a Traditional IDE

C++ code is edited in the developer's editor of choice (which may integrate a version control system). Graph nodes are (re)generated using the *nodegen* tool, which can optionally be integrated into a Makefile or CMake-based build system: see Tutorial 9: Developer Workflow. The advantage of this approach is that it allows the developer to use the work environment he is accustomed to, with all the advantages of an IDE that *kpedit* does not offer.

One approach does not exclude the other; they can be combined.

4.7 Configuration

Whenever a Kirpi application such as *kpedit* or *nodegen* is run, or a Kirpi-based plug-in is loaded into another application, a Kirpi configuration file is loaded. The file contains a number of settings, most of which are optional. For example:

ROOT_DIR

> *Required.* The path of the directory where Kirpi resource files are installed (for the location of the include directory, in particular).

NODE_PATH

> *Optional.* Directories where Kirpi searches for graph node files (in addition to the directory containing the graph). This makes it possible to identify nodes in a graph file by name, rather than with a complete path. Also, you can then move a graph file to a different directory without having to change the paths of the node files it references.

For details about the configuration file syntax and about all the settings it may contain, see the sample `kirpi.conf` file installed with Kirpi.

4.8 Environment Variables

The following environment variables can be defined to influence the behavior of Kirpi applications and plug-ins.

KIRPI_CONFIG_FILE

> By default, Kirpi looks for a configuration file called `kirpi.conf` (see Configuration above) first in the directory where the application or plug-in is located, and then, if it is not found there, in the user's home directory. But the KIRPI_CONFIG_FILE environment variable can be defined to specify a different path for the file to be loaded.
>
> For details about the Kirpi configuration file, see the default configuration file provided.

KIRPI_DEBUG

When Kirpi compiles graphs it may generate a number of temporary C++ files that are automatically cleaned up afterwards. But if the KIRPI_DEBUG environment variable is set to anything but "0", those temporary files are left on disk, and Kirpi writes messages to standard output indicating the paths of the files and what they contain.

KIRPI_FORCE_COMPILE

If set to anything other than "0", existing bitcode files are not loaded from disk; they are saved, however. This may be useful to force nodes and graphs to be recompiled, if for any reason their bitcode files may be out of date.

KIRPI_FORCE_OPTIM_FLAGS

Set this to "-O" or to "-g", for example, to force all nodes and graphs to be compiled with the given optimization flags, regardless of the compiler options specified in their files. Setting this variable has several implications:

- Bitcode files are not loaded from disk; all graphs and nodes are recompiled.
- Generated bitcode is not saved to disk.
- "-O" and "-g" compiler options given in node and graph files are ignored.

The variable can contain several options separated by spaces.

KIRPI_LICENSE_FILE

By default, Kirpi looks for a license file called `kirpi.lic` in the `kirpi` directory where Kirpi was installed (see Installing Kirpi). But the KIRPI_LICENSE_FILE environment variable can be defined to specify a different path for the file to be loaded.

KIRPI_PROFILING

Profiling of Kirpi graphs and applications is disabled by default, but can be activated by setting the KIRPI_PROFILING environment variable to anything but "0".

5 Examples

Kirpi comes with the source code for several use case examples: a standalone application (*kpexec*) and three plug-ins for Maya. All come precompiled as well: the *kpexec* binary can be found in the `bin` directory, and the Maya plug-ins (shared libraries and MEL scripts) can be found in the `maya` directory. The `examples/maya` directory also contains a shell script to compile the plug-ins, which can be used as a model for developing and building other plug-ins. The Maya plug-ins provided with Kirpi have been compiled for Maya 2020 or Maya 2022, depending on the version of Kirpi that was installed.

5.1 kpexec

kpexec is a simple example of a standalone application that uses libkirpi to load, compile and execute Kirpi graphs. It loads a graph into memory, then reads commands from standard input until a "quit" command or EOF is reached. Commands include "compile", "execute", "set" (to set input values) and "get" (to retrieve output values). For details on how to use *kpexec*, see `doc/kpexec.rst`.

The source code for *kpexec* in `examples/kpexec` should be largely self-explanatory, but we will give a brief overview here of the libkirpi classes and functions it uses.

Note that all libkirpi classes are defined in the `kirpi` namespace. The source code for *kpexec* includes a **using** directive so that the namespace need not be given explicitly, and in the explanation that follows, the namespace is omitted.

Loading a Graph

The most important class provided by libkirpi (and practically the only class used by *kpexec*) is `Graph`. To load a graph into memory, use the `Graph::load()` method, which returns a pointer to a `Graph`. *kpexec* uses it as follows:

```
/*
 * Loads a graph from a file, returns it and the options needed for linking
 * (library_options) and for compiling it (other_options).
 */
static Graph* loadGraph(
    const std::string& filename,
    std::vector<std::string>& library_options,
    std::vector<std::string>& other_options)
{
    Graph *graph = Graph::load(filename);
    if (!graph) {
        return nullptr;
    }

    std::vector<std::string> include_options;
    library_options.clear();
    other_options.clear();
```

```
        graph->findCompilerOptions(
            include_options, library_options, other_options);
        other_options.insert(
            other_options.end(), include_options.begin(),
            include_options.end());

        return graph;
    }
```

The above code loads the graph and, if successful, calls `Graph::findCompilerOptions()` to retrieve the options that will be needed to compile the graph (see below).

Compiling a Graph

Before a graph can be executed it must be compiled. In libkirpi this is done in two steps. First, any external shared libraries (DSOs on Linux) used by the graph must be loaded into memory. Then, when the graph is compiled, the JIT compiler will search those DSOs for any functions and variables that the graph uses but are not defined by the graph or by a static library it links with.

The first step (loading external libraries) is done by `Graph::loadLibraries()`; the second (compiling and linking) is done by `Graph::compile()`. *kpexec* compiles like so:

```
    if (graph->loadLibraries(library_options, static_libraries)
        && graph->compile(false, other_options, static_libraries)) {
        std::cout << "compilation OK" << std::endl;
    }
```

The compiler options were retrieved when the graph was loaded: see `Graph::findCompilerOptions()`. libkirpi separates compiler options into three categories -- include options (such as `-I`), library options (such as `-l` and `-L`) and all others. But for our purposes we only need to distinguish between library options (for loading external libraries) and all the others.

The reason compilation in libkirpi is done in two separate steps like this is that an application might want to compile the graph in a separate thread, to avoid blocking the user interface while it compiles. But DSOs must be loaded from the main thread; only `Graph::compile()` can be invoked from a different thread.

External static libraries (`.a` files on Linux) are detected by `Graph::loadLibraries()` but are not loaded into memory immediately the way shared libraries are. Instead they are passed on to `Graph::compile()`, which is responsible for loading them and linking them with graph code.

There is also another important difference between shared and static libraries used by the graph. Once a shared library is loaded into memory, it is made available to *all* graphs, and it is never unloaded; it remains in memory until the application exits. As a consequence, if a shared library is recompiled, changes made to the library will not be taken into account until the application is restarted. A static library, on the other hand, is made available only to the graph that explicitly links with it, and is unloaded when the graph is.

Note for Windows

The Windows version of Kirpi does not currently support JIT compilation. Instead, the bitcode for graphs and nodes is compiled in a separate process to DLLs which are then loaded into memory. This is mostly transparent to the user, and makes no difference when programming with the API. As a result, however, calls to `Graph::loadLibraries()` do not actually do anything on Windows.

Executing a Graph

Once the graph has been compiled, it is ready to execute by calling `Graph::execute()`:

```
if (graph->loadLibraries(library_options, static_libraries)
    && graph->execute(false, other_options, static_libraries)) {
    std::cout << "execution OK" << std::endl;
}
```

This looks a lot like the compilation step, because `Graph::execute()` actually compiles the graph, too, if necessary, so it needs all the same parameters as `Graph::compile()`. To determine whether the graph actually needs compiling you can call `Graph::getCompileNeeded()`. If that method returns true, then `Graph::execute()` will compile the graph first.

Setting Inputs and Retrieving Outputs

Executing a graph is generally more useful if you set its input values beforehand and retrieve its output values afterwards. This is done by the methods `Graph::setInput()` and `Graph::getOutput()`. Both methods take the name of a plug (an input or an output defined by the graph) and a reference to a value. For example:

```
graph->setInput("count", 3)
```

or:

```
float answer;
graph->getOutput("result", answer);
```

Note that these methods are actually templates: the second parameter must be a reference to a value of the same type as that of the corresponding plug. In the first example above, it is assumed that "count" is an input of type `int`; in the second example, that "result" is an output of type `float`. If the parameter does not correspond to the type of the named plug, both methods return an error.

Note also that `Graph::setInput()` and `Graph::getOutput()` access variables that are created in memory when the graph is compiled. This has two important consequences:

- They cannot be called until the graph has been compiled.

- If the graph is recompiled, any changes made to the inputs are lost, as are any outputs previously computed by executing the graph. Inputs and outputs are reset to their default values when the graph is compiled.

5.2 Maya Deformer

kpDeformer is an example of a Maya plug-in that implements a custom deformer node. The deformer node uses a Kirpi graph to modify the vertices of a mesh object. Its `deform()` method passes the mesh object to the graph, executes the graph and retrieves an output containing the modified vertex positions.

The `deformer` directory also contains a very simple Maya scene, `scene.ma`, and the Kirpi graph file that it uses, `kpDeformer.graph`. The graph uses an image file to displace the vertices of a mesh in the direction of their normal vectors.

Installing the Plug-in

For the plug-in to work, Maya needs to find three files:

- `kpDeformer.so` (on Linux) or `kpDeformer.mll` (on Windows), the compiled plug-in.
- `AEkpDeformerTemplate.mel`, a MEL script provided with this example.
- `AEkpNodeCommon.mel`, a MEL script provided by the kpmaya library.

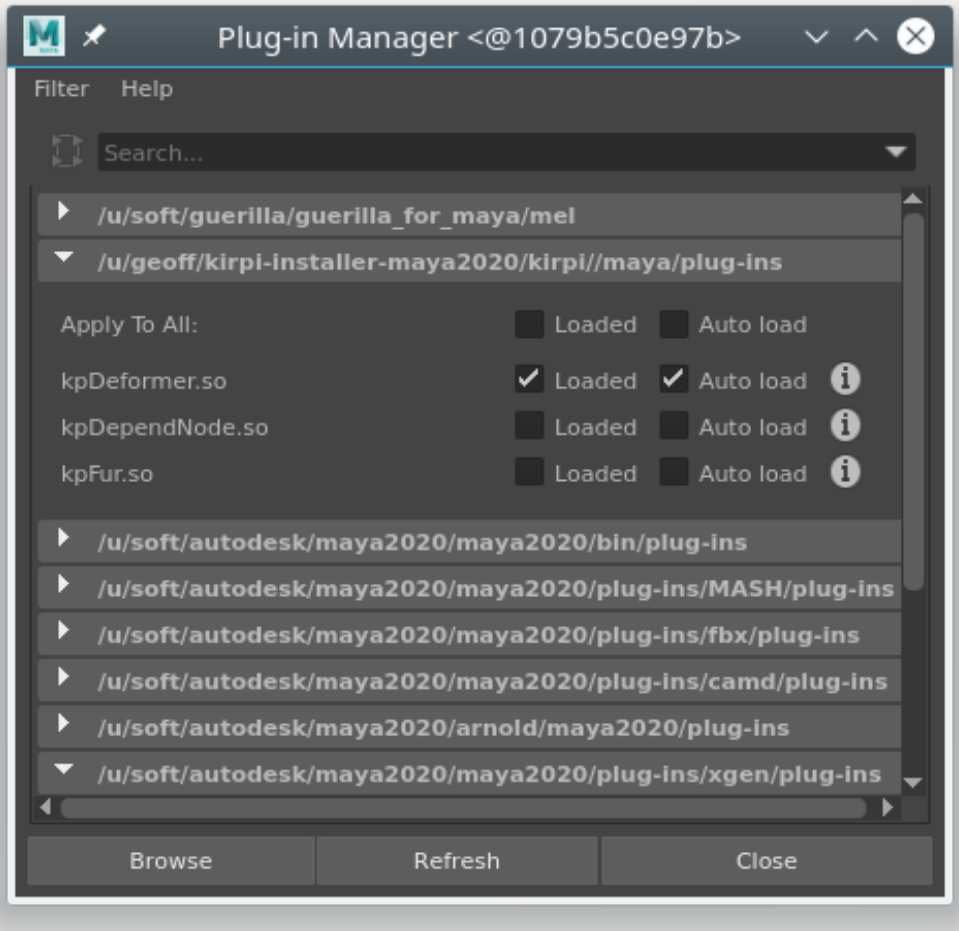
The simplest way to install the plug-in is to copy (or link) the plug-in to the directory defined by MAYA_PLUG_IN_PATH, and to copy (or link) the two MEL scripts to the directory defined by MAYA_SCRIPT_PATH. For details about installing Maya plug-ins, see the Maya documentation.

Note that the compiled plug-in and MEL scripts are installed with Kirpi, in the `maya` directory at the root of the installation.

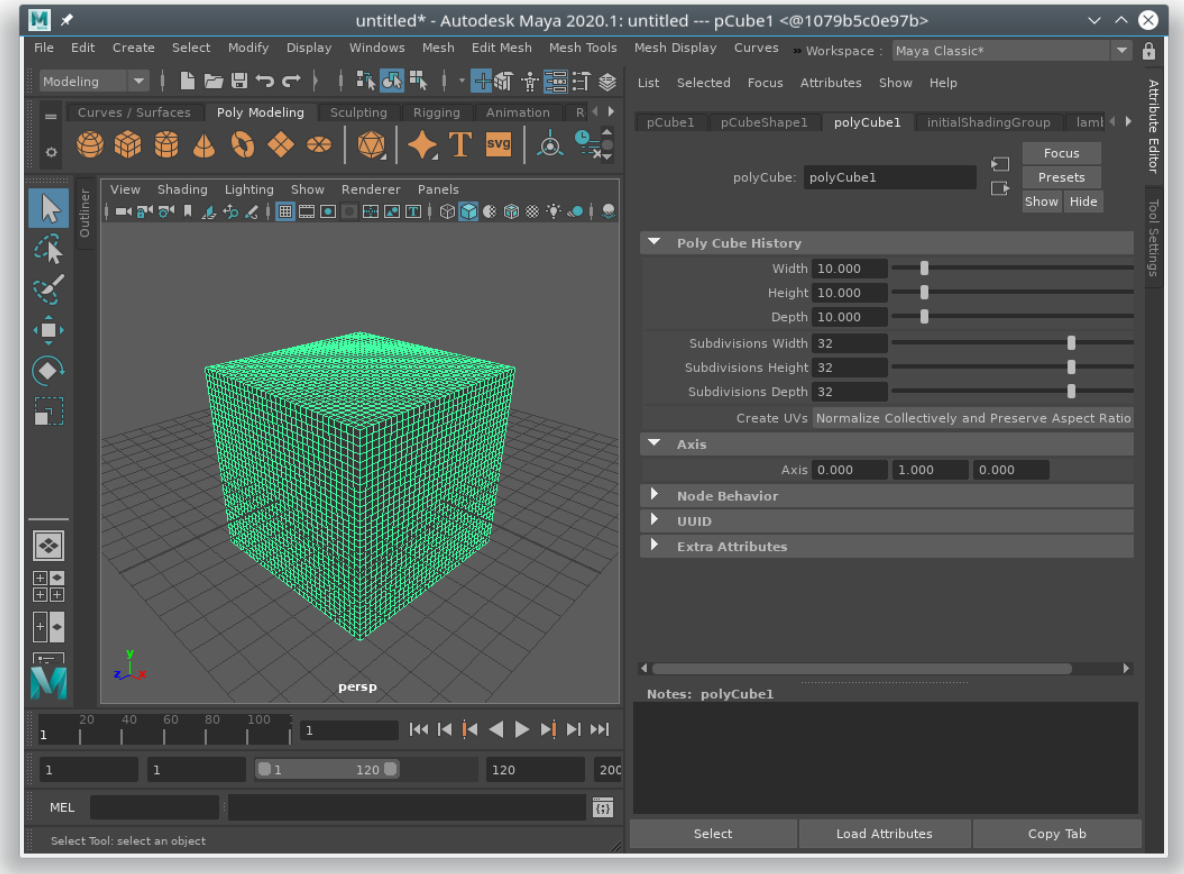
Once installed, the plug-in needs to find `kirpi.conf`, which may be located either in the user's home directory or in the same directory as the plug-in. Alternatively, the KIRPI_CONFIG_FILE environment variable may be set to the path of the configuration file.

Using the Plug-in

1. Load the plug-in in Maya. Open the plug-in manager from the menu bar (*Windows → Settings/Preferences → Plug-in Manager*). Find `kpDeformer.so` or `kpDeformer.mll` in the plug-in manager window and ensure it is loaded.

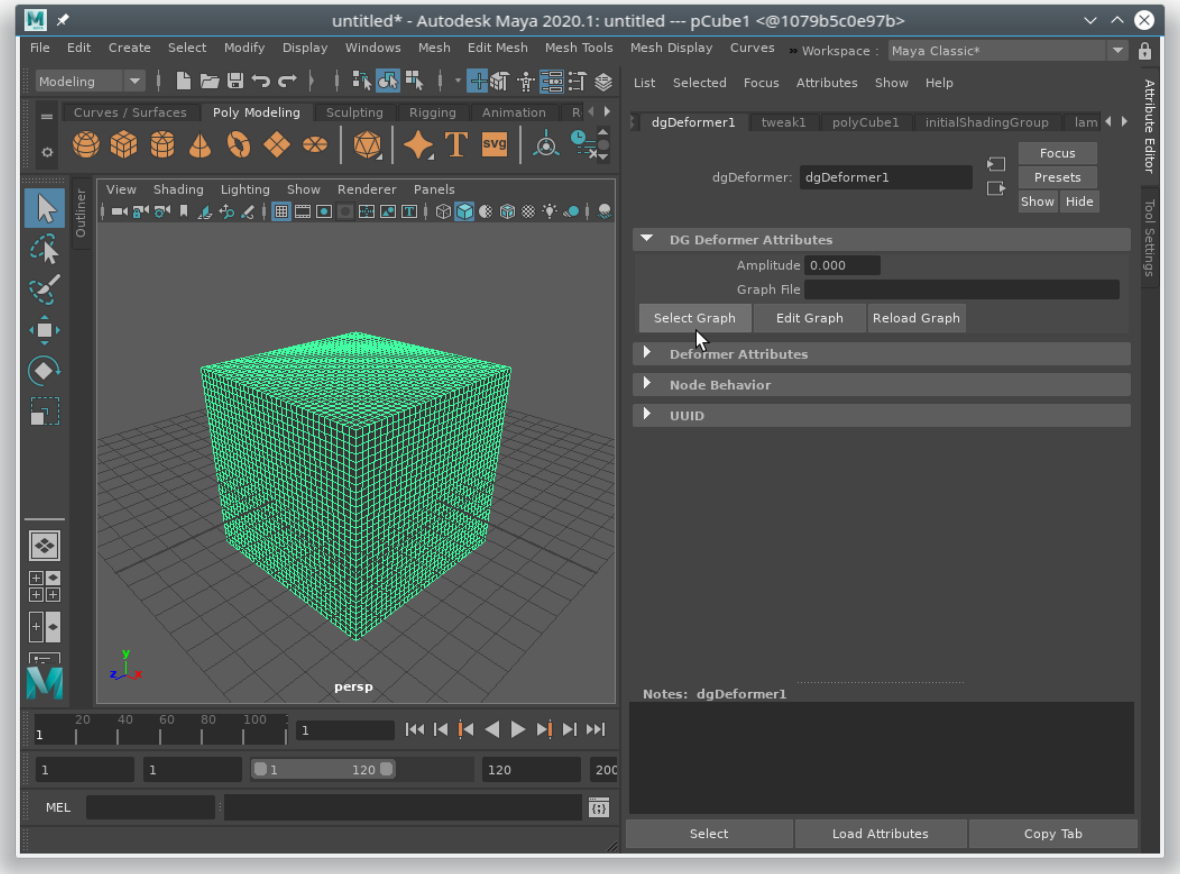

Plug-in Manager <@1079b5c0e97b>
Filter
Help
Search...
/u/soft/guerilla/guerilla_for_maya/mel
/u/geoff/kirpi-installer-maya2020/kirpi//maya/plug-ins
Apply To All:
Loaded
Auto load
kpDeformer.so
Loaded
Auto load
kpDependNode.so
Loaded
Auto load
kpFur.so
Loaded
Auto load
/u/soft/autodesk/maya2020/maya2020/bin/plug-ins
/u/soft/autodesk/maya2020/maya2020/plug-ins/MASH/plug-ins
/u/soft/autodesk/maya2020/maya2020/plug-ins/fbx/plug-ins
/u/soft/autodesk/maya2020/maya2020/plug-ins/camd/plug-ins
/u/soft/autodesk/maya2020/arnold/maya2020/plug-ins
/u/soft/autodesk/maya2020/maya2020/plug-ins/xgen/plug-ins
Browse
Refresh
Close

2. Create a mesh object with UV coordinates (a cube, for example). Make sure the object is subdivided enough that it has lots of vertices to deform.

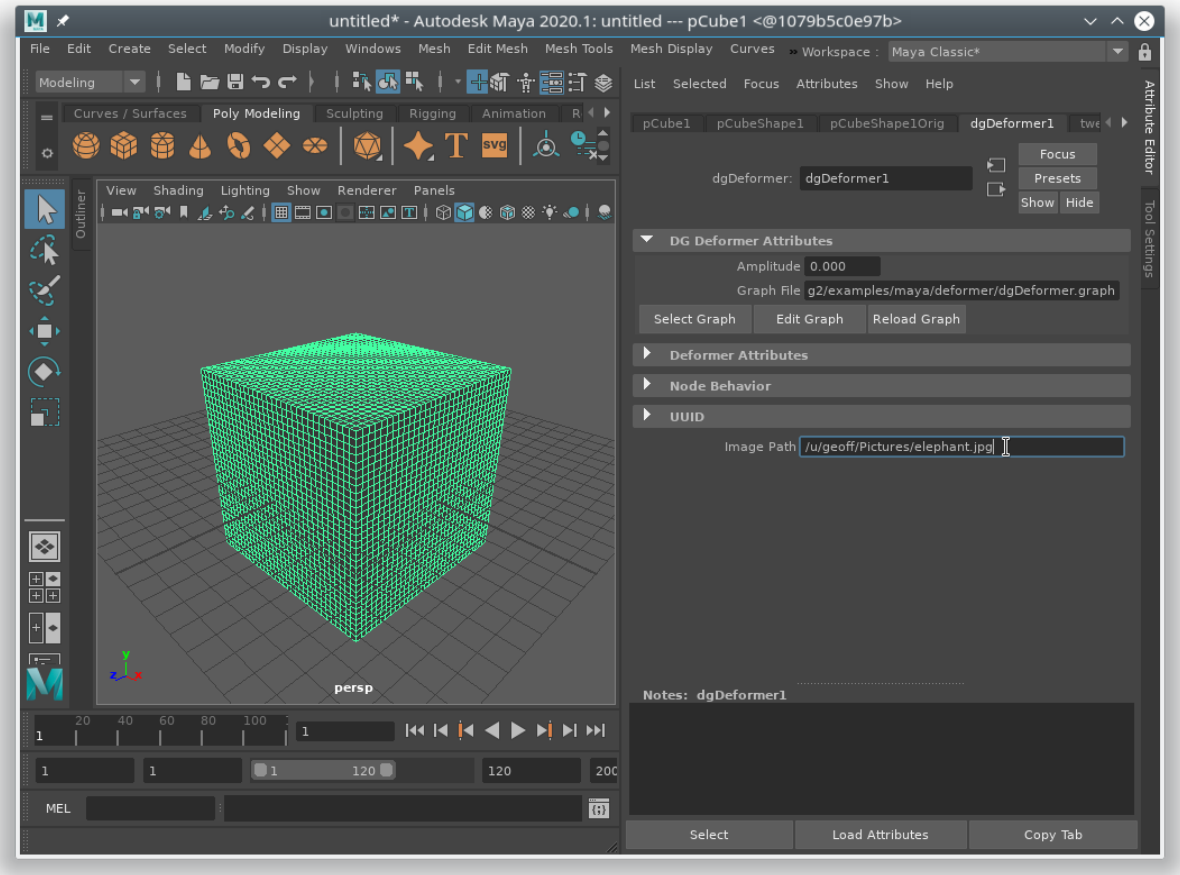


3. Create a *kpDeformer* node by entering the MEL command `deformer -type kpDeformer`. The deformer is attached to the selected mesh object.

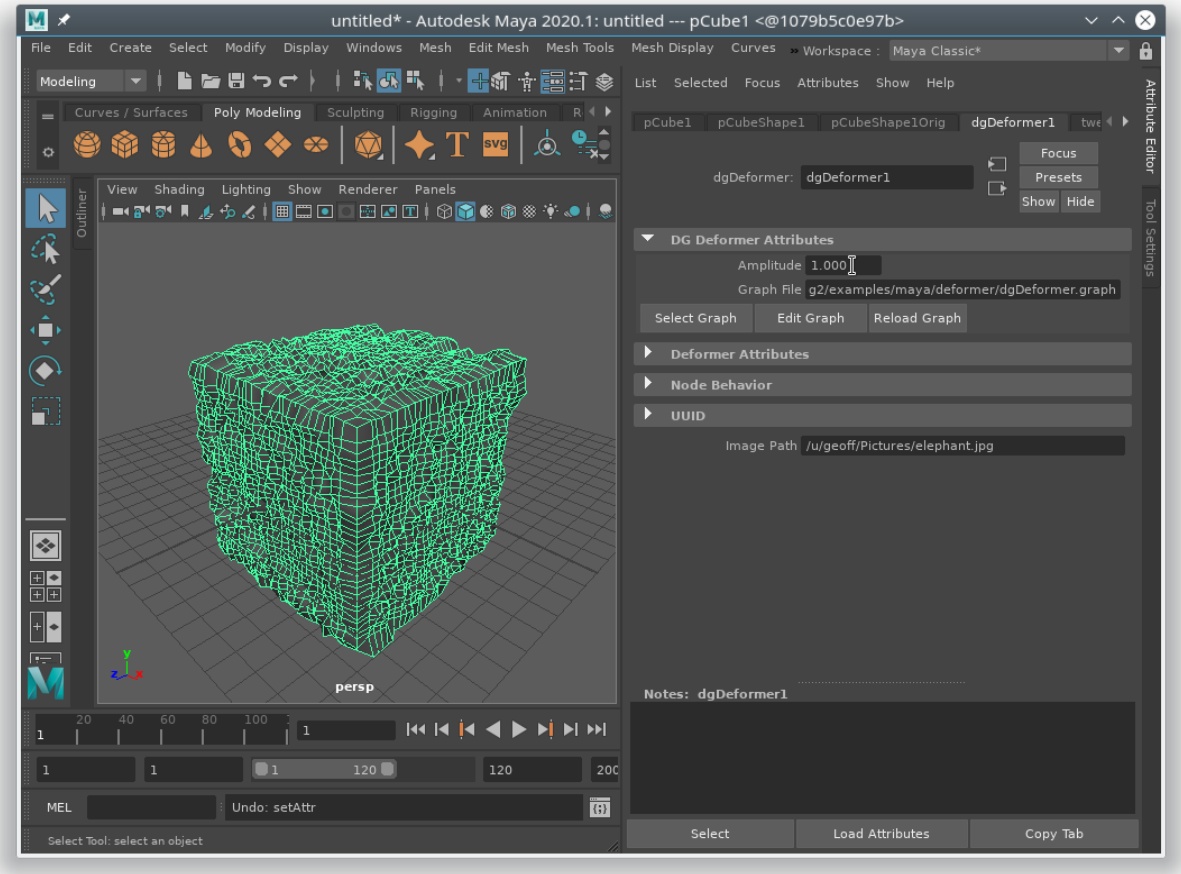
4. Point the *kpDeformer* to the graph file provided with this example. Select the deformer node in the attribute editor, click on the *Select Graph* button and select the file `kpDeformer.graph`. The deformer will then compile the graph and add a new attribute called *Image Path*.



5. Enter the path of an image file in the *Image Path* attribute.



6. Set the *amplitude* attribute of the deformer to 1 (or to any positive value). The object's vertices should be displaced in the direction of their normals.



7. Click on the *Edit Graph* button to open the graph, and edit it to perform whatever computations you want on the mesh vertices.

You can add additional inputs to the graph, if necessary. If a new input's type is valid for a Maya attribute (*e.g.* int, double, MString, etc.), a Maya attribute with the same name is added to the deformer node. The user can then set its value from Maya's Attribute Editor, or connect it to the output of another node in Maya's Node Editor (*Windows → Node Editor*).

You can also add an input to the graph by dragging and dropping a DAG node from Maya's Outliner into the Kirpi graph editor, using the middle mouse button. In that case, Kirpi creates an input of type MObject whose name depends on the type of the DAG node ("transform", for example). When the graph is evaluated, that input is set to the corresponding DAG node; whenever the DAG node is marked dirty by Maya, the Kirpi deformer node is marked dirty as well, causing the graph to be reevaluated when needed.

8. Save the modified graph to a different graph file, to avoid overwriting the example graph, by selecting *File → Save As...* in the graph editor's menu bar. The path to the graph file is saved in the Maya scene.

Advanced Usage

In some cases, it may be useful for the graph to know which of its inputs have changed since the last time the graph was evaluated. The graph might cache the results of expensive calculations and reuse them next time, for example, if some inputs have not changed. This can be done by adding an input to the graph called "dirtyPlugs".

If the graph has an input called "dirtyPlugs", then before Kirpi evaluates the graph, it sets the "dirtyPlugs" input to the names of the input plugs that Maya has marked dirty. Its type must be *std::set<std::string>*.

5.3 Maya Fur

kpFur is an example of a Maya plug-in that implements a custom shape node. The shape node uses a Kirpi graph to generate hairs (curves) on the surface of a mesh. The graph takes as input an emitter mesh; its output is a vector of Curve instances (Curve is a class defined by the fur library, also provided). The fur library and a number of example Maya scenes, plus the Kirpi graph files they use, can all be found in the `examples/maya/fur` directory.

Installing the Plug-in

For the plug-in to work, Maya needs to find three files:

- `kpFur.so` (on Linux) or `kpFur.mll` (on Windows), the compiled plug-in.
- `AEkpFurTemplate.mel`, a MEL script provided with this example.
- `AEkpNodeCommon.mel`, a MEL script provided by the kpmaya library.

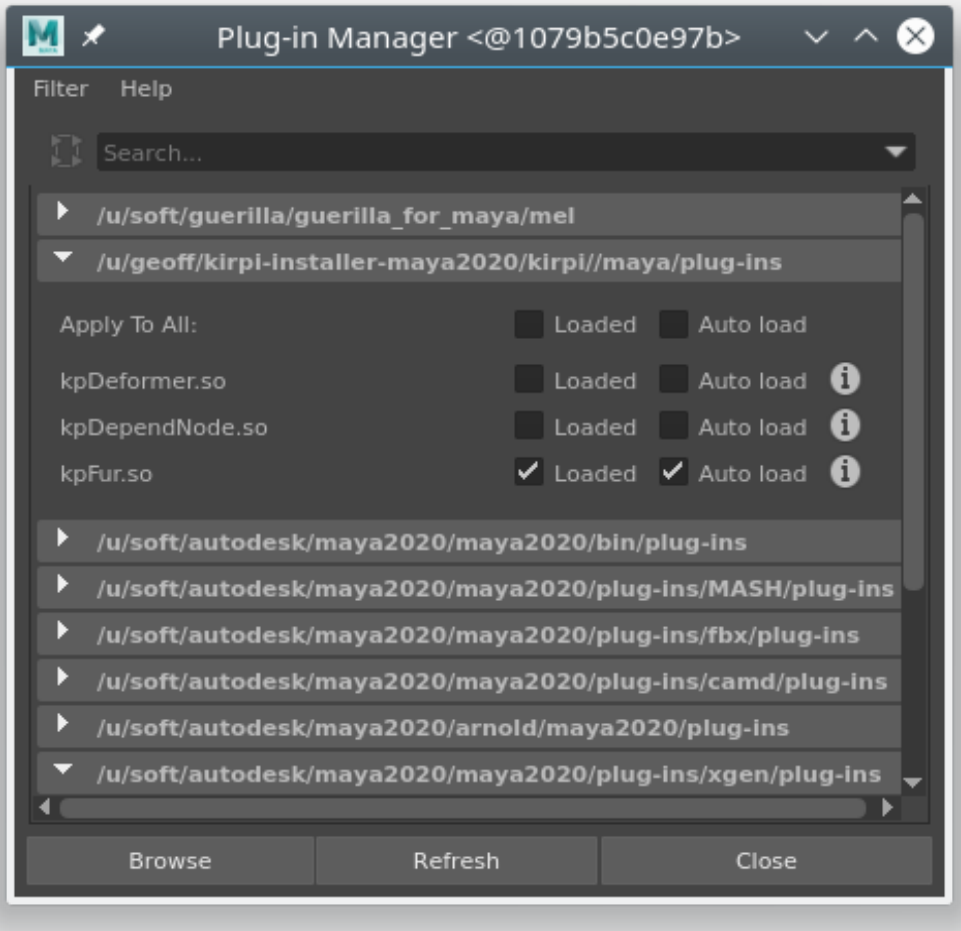
The simplest way to install the plug-in is to copy (or link) the plug-in to the directory defined by MAYA_PLUG_IN_PATH, and to copy (or link) the two MEL scripts to the directory defined by MAYA_SCRIPT_PATH. For details about installing Maya plug-ins, see the Maya documentation.

Note that the compiled plug-in and MEL scripts are installed with Kirpi, in the `maya` directory at the root of the installation.

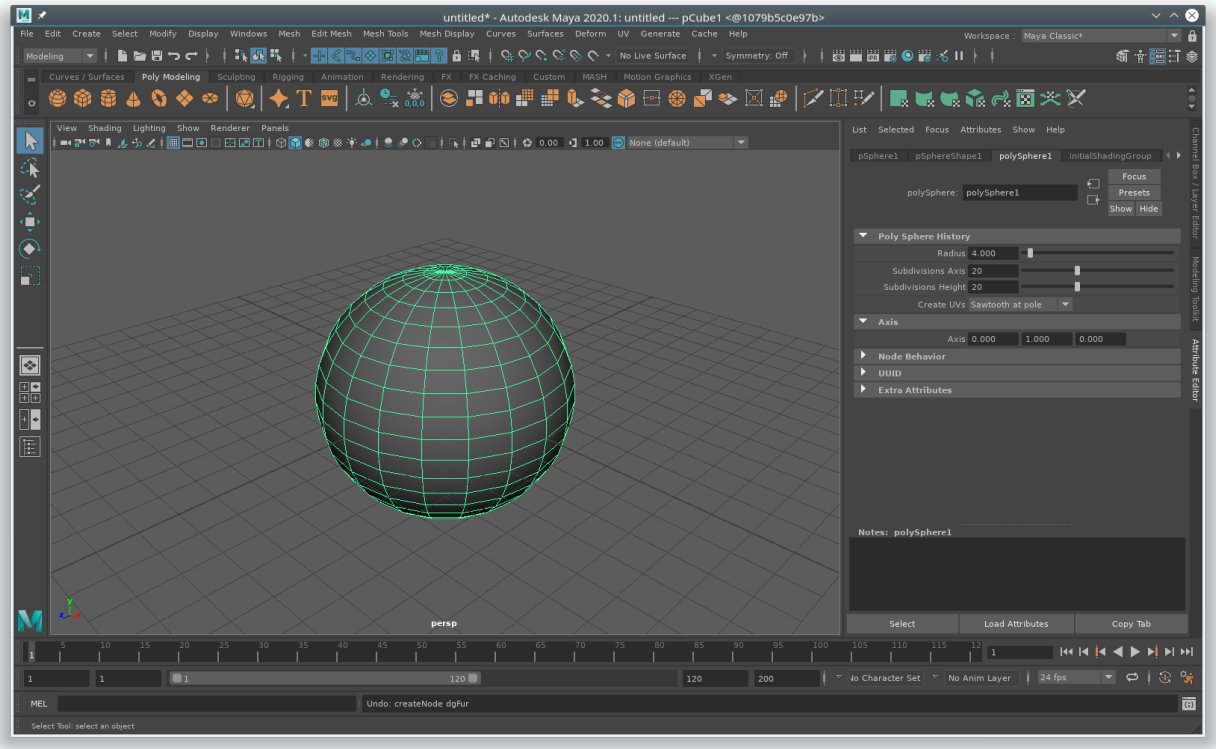
Once installed, the plug-in needs to find `kirpi.conf`, which may be located either in the user's home directory or in the same directory as the plug-in. Alternatively, the KIRPI_CONFIG_FILE environment variable may be set to the path of the configuration file.

Using the Plug-in

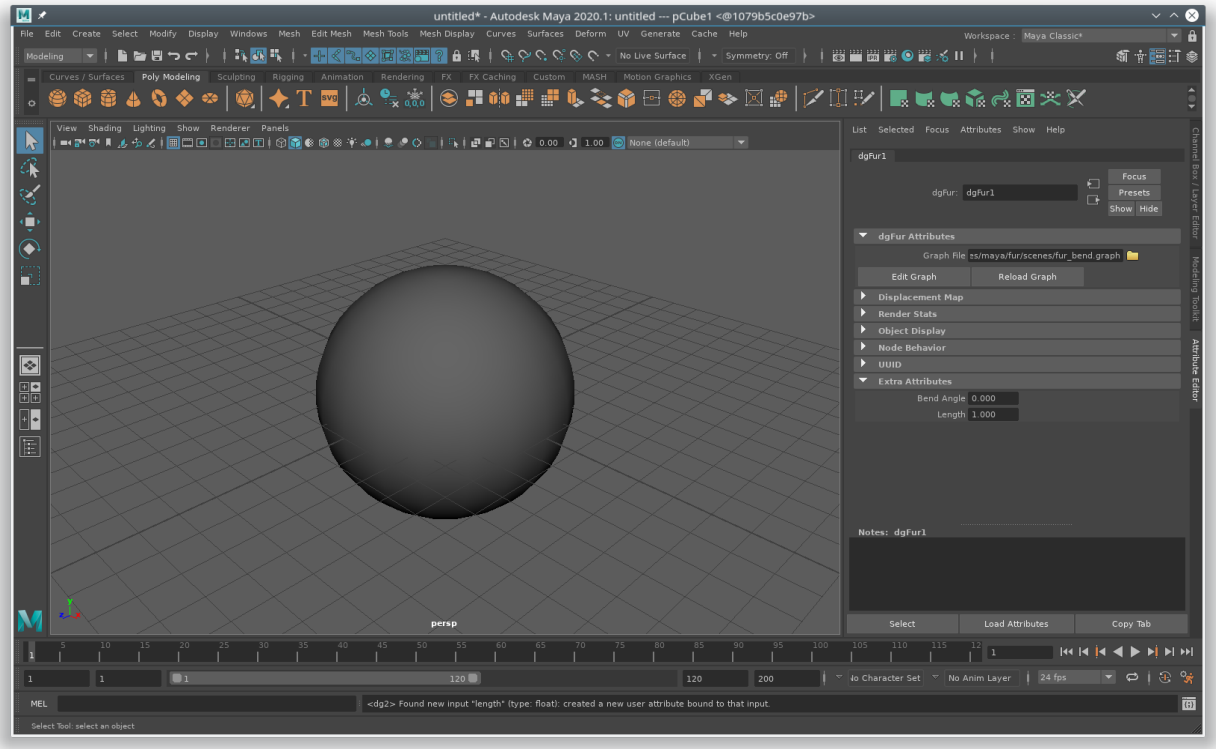
1. Load the plug-in in Maya. Open the plug-in manager from the menu bar (*Windows → Settings/Preferences → Plug-in Manager*). Find `kpFur.so` or `kpFur.mll` in the plug-in manager window and ensure it is loaded.



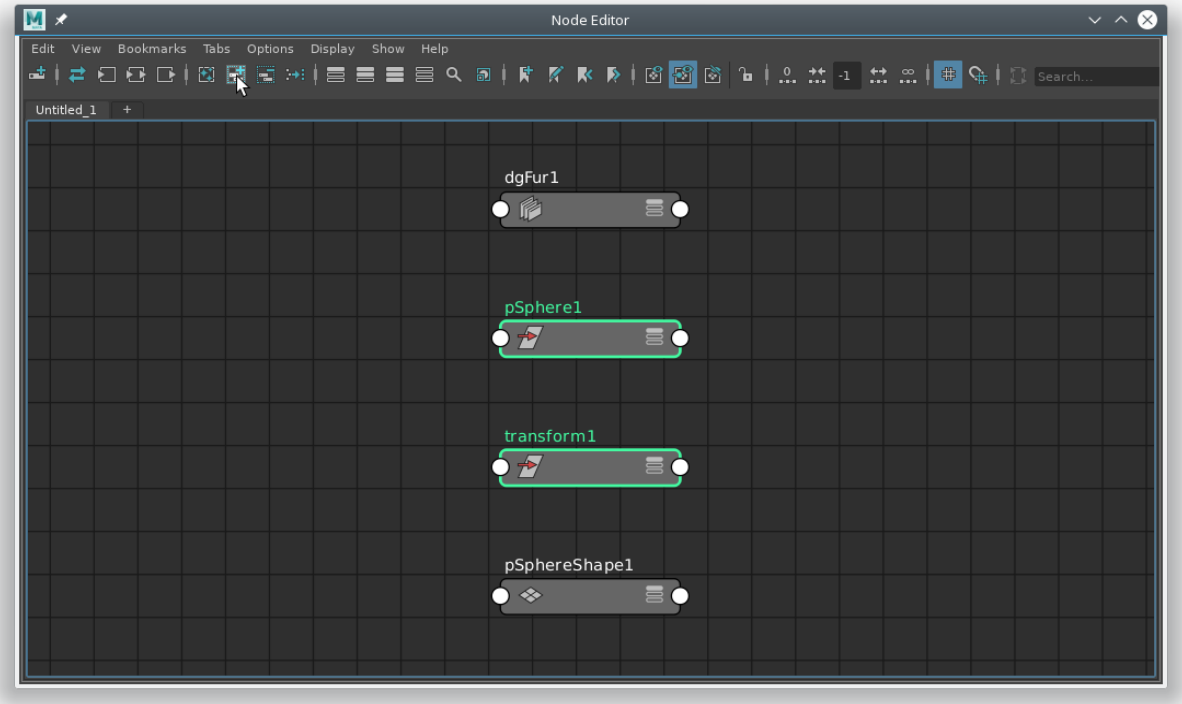
2. Create a mesh object (in this example a sphere of radius 4). The fur shape will emit hairs at each of the mesh's vertices.



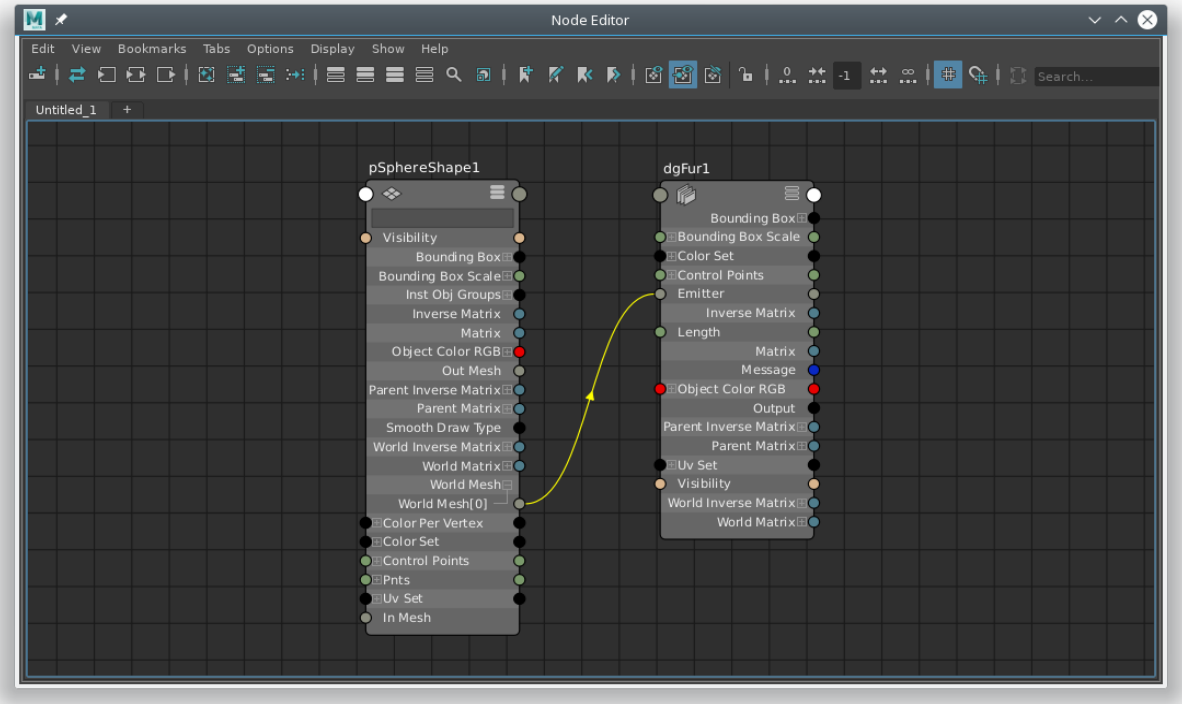
3. Create a *kpFur* node by entering the MEL command `createNode kpFur`. In the Attribute Editor, click on the folder button to the right of the node's *Graph File* attribute to bring up a file browser. In the browser, navigate to the folder containing the fur example graphs and pick one (`fur_bend.graph` here).



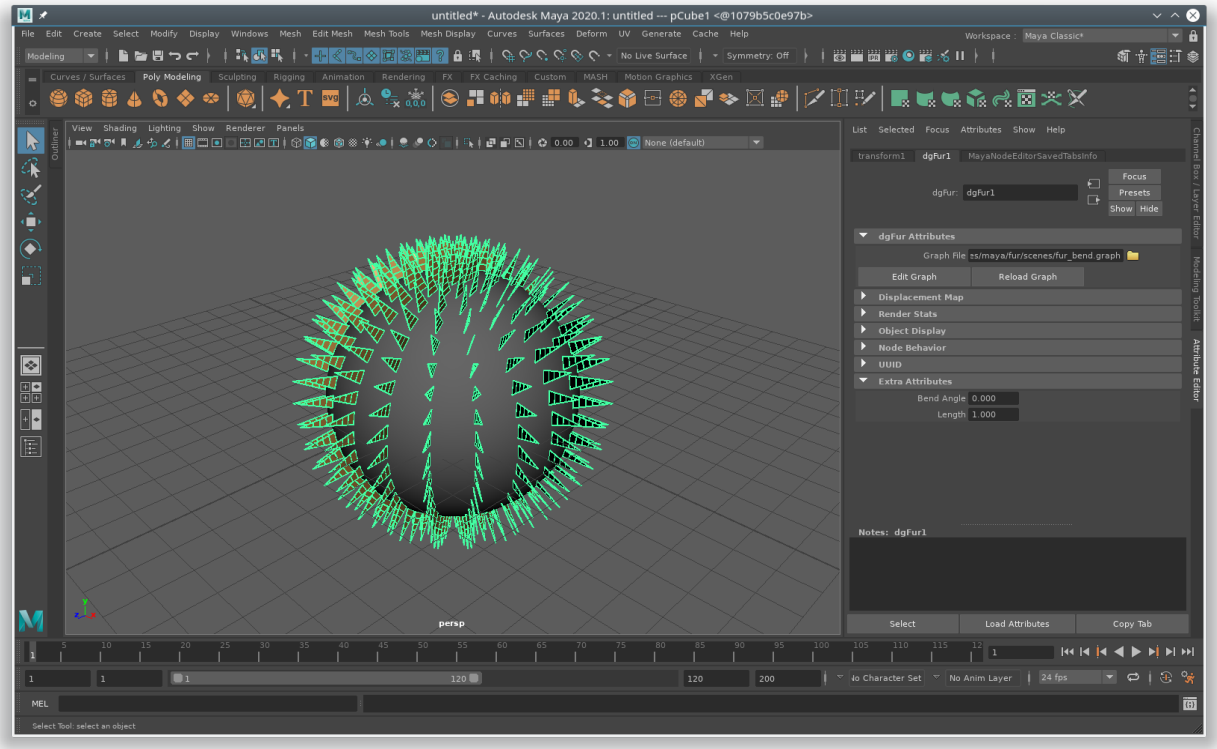
4. Open Maya's Node Editor (*Windows → Node Editor*). Make sure both the emitter shape node (the sphere) and the fur node are visible in it. The simplest way to do that is to select both in the Outliner, then to click on the Node Editor button *Add selected nodes to graph*.



5. In the Node Editor, move the fur node to the right of the emitter node. Click several times on the bar icon in the upper right corner of each node until all of the nodes' attributes are displayed. Then, connect the emitter node's *World Mesh* output to the fur node's *emitter* input.



6. Close the Node Editor. In the 3D view, hairs have now been emitted at the emitter shape's vertices.



7. Play with the fur node's attributes (*Bend Angle* and *Length* in this case). Click on the *Edit Graph* button to see and/or modify the graph. If you wish to save the modified graph, save to a different file name, to avoid overwriting the example graph, by clicking on *File → Save As...* in the graph editor menu bar. The path of the graph file is saved with the Maya scene.

Advanced Usage

In some cases, it may be useful for the graph to know which of its inputs have changed since the last time the graph was evaluated. The graph might cache the results of expensive calculations and reuse them next time, for example, if some inputs have not changed. This can be done by adding an input to the graph called "dirtyPlugs".

If the graph has an input called "dirtyPlugs", then before Kirpi evaluates the graph, it sets the "dirtyPlugs" input to the names of the input plugs that Maya has marked dirty. Its type must be *std::set<std::string>*.

5.4 Maya Dependency Node

kpDependNode is an example of a Maya plug-in that implements a custom dependency graph node. The node contains a Kirpi graph that can be edited by the user and can perform any sort of computation.

Out of the box, a *kpDependNode* does not actually do anything. It is the user's responsibility to edit the Kirpi graph, to make it read some input values and generate output values. As inputs and outputs are added to the Kirpi graph, Maya attributes with the same names are added automatically to the Maya dependency node. The user can then use Maya's Node Editor to connect the node's outputs, and perhaps its inputs, to other plugs in the Maya scene's dependency graph.

A very simple Maya scene, and the Kirpi graph file that it uses, are also provided in the `examples/maya/depend_node` directory. The sample graph just adds two input values; the output value is plugged into the radius attribute of a torus shape.

Installing the Plug-in

For the plug-in to work, Maya needs to find three files:

- `kpDependNode.so` (on Linux) or `kpDependNode.mll` (on Windows), the compiled plug-in.
- `AEkpDependNodeTemplate.mel`, a MEL script provided with this example.
- `AEkpNodeCommon.mel`, a MEL script provided by the kpmaya library.

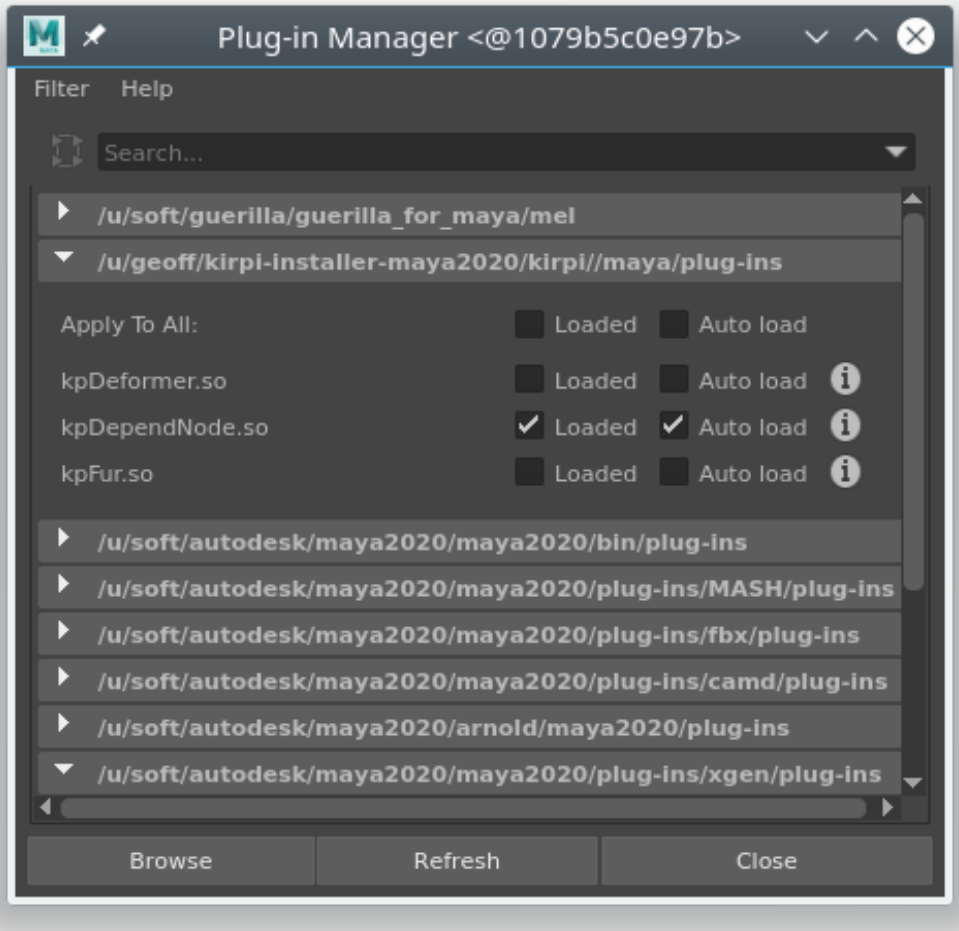
Note that the compiled plug-in and MEL scripts are installed with Kirpi, in the `maya` directory at the root of the installation.

The simplest way to install the plug-in is to copy (or link) the plug-in to the directory defined by MAYA_PLUG_IN_PATH, and to copy (or link) the two MEL scripts to the directory defined by MAYA_SCRIPT_PATH. For details about installing Maya plug-ins, see the Maya documentation.

Once installed, the plug-in needs to find `kirpi.conf`, which may be located either in the user's home directory or in the same directory as the plug-in. Alternatively, the KIRPI_CONFIG_FILE environment variable may be set to the path of the configuration file.

Using the Plug-in

1. Load the plug-in in Maya. Open the plug-in manager from the menu bar (*Windows → Settings/Preferences → Plug-in Manager*). Find `kpDependNode.so` or `kpDependNode.mll` in the plug-in manager window and ensure it is loaded.



2. Create a *kpDependNode* by entering the MEL command `createNode kpDependNode`. The new node has no inputs, no outputs and an empty graph.

3. In Maya's Attribute Editor, click on the *Edit Graph* button to open the (empty) Kirpi graph. Add nodes to the graph, and add appropriate inputs and outputs for whatever the graph is supposed to compute. When an input or an output is added to the graph, if its type is valid for a Maya attribute (*e.g.* int, double, MString, etc.), a Maya attribute with the same name is added to the dependency graph node. The user can then set the values in the Attribute Editor, or connect inputs and outputs to other dependency graph nodes in the Node Editor (*Windows → Node Editor*).

You can also add an input to the graph by dragging and dropping a DAG node from Maya's Outliner into the Kirpi graph editor, using the middle mouse button. In that case, Kirpi creates an input of type MObject whose name depends on the type of the DAG node ("transform", for example). When

the graph is evaluated, that input is set to the corresponding DAG node; whenever the DAG node is marked dirty by Maya, the Kirpi dependency node is marked dirty as well, causing the graph to be reevaluated when needed.

4. Save the graph to a file by selecting *File → Save As...* from the graph editor's menu bar. The path to the graph file is saved in the Maya scene.

Advanced Usage

In some cases, it may be useful for the graph to know which of its inputs have changed since the last time the graph was evaluated. The graph might cache the results of expensive calculations and reuse them next time, for example, if some inputs have not changed. This can be done by adding an input to the graph called "dirtyPlugs".

If the graph has an input called "dirtyPlugs", then before Kirpi evaluates the graph, it sets the "dirtyPlugs" input to the names of the input plugs that Maya has marked dirty. Its type must be *std::set<std::string>*.

Exporting as a Unique Maya Plug-in

If you put a lot of work into developing a useful *kpDependNode*, you may want to share it with other users, including users who do not have access to Kirpi (even though Kirpi is free to use). For that scenario, Kirpi has an experimental feature which allows you to export all the C++ code needed to build a Maya plug-in with its own name and unique ID, and which can be used without installing Kirpi.

To take advantage of this feature, select *File → Export as Maya Plug-in...* from the Kirpi graph editor's menu bar. A dialog box prompts you for information such as the name of the new plug-in, its unique ID (to be obtained from Autodesk), and the directory where you want to save the plug-in. When you click on OK, Kirpi writes four files to the given directory:

- *name*`_graph.cpp`
- *name*`_plugin.cpp`
- `CMakeLists.txt`
- `README.txt`

where *name* is the plug-in name provided. The `README.txt` file contains instructions for building the new plug-in.

Please note that a couple features of *kpDependNode* are not supported when a plug-in is exported this way. First, any graph inputs which were created by dragging and dropping a Maya DAG node onto the graph will not be set. And second, the "dirtyPlugs" input, if present, will not be set.

5.5 Python Integration

This example demonstrates how to integrate Kirpi into a Python-based workflow. The idea is to use a Kirpi graph in a Python application, as if it were native Python code. This strategy combines the ease of use of Python with the power of C++.

Python Binding

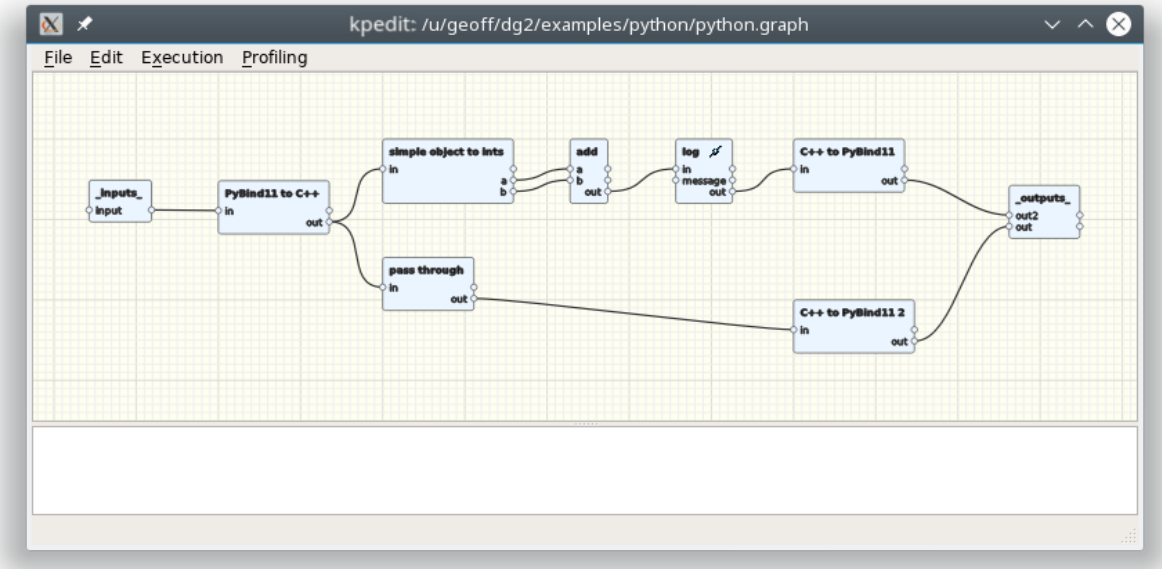
We have implemented a partial Python binding of the libkirpi library, using pybind11 (we could just as easily have used Boost.Python). The binding is located in the file `kirpi_binding.cpp`, and contains just a small subset of methods from the `Graph` class allowing you to load a graph, compile it, set its inputs, execute it, and retrieve its outputs.

The Python binding itself is not part of Kirpi; for more information about creating Python bindings with pybind11, see the pybind11 documentation. There are a couple points worth noting about the Kirpi bindings, however:

- We derive a subclass of `pybind11::object`, called `kpPythonObject`, because pybind11 makes instances of `pybind11::object` invisible outside of the module where they are defined, and we need for those instances to be visible.
- We register the `kpPythonObject` type with Kirpi so that graph inputs and outputs can be instances of `kpPythonObject`. We do that by calling the `KIRPI_REGISTER_PLUG_TYPE()` macro defined by the `Graph` class.

Test Script

We are going to use the Python binding to load the following graph file, compile it and execute it:



The input to our graph is an instance of a C++ class called `simple`, defined below. The `simple` class also has a Python binding, so that it is accessible in both worlds. We create the `simple` object in the Python script and pass it to the Kirpi graph.

The first graph node, PyBind11toC++, converts the input Python object into a C++ object. Its output is a C++ object as in all the above tutorials.

The branch of the graph connected to the *out2* output extracts the values contained by the `simple` object, adds them, displays the result and converts the result to a pybind11 object.

The branch of the graph connected to the *out* output just converts the Python object to C++ and back again, demonstrating how a more complex Python object can be generated by a graph.

Note that this graph cannot be executed in *kpedit*, because its inputs and outputs are of Python types not found in *kpedit*.

The `simple` class is defined as follows:

```
class simple {
public:
    simple()
        : a(0),
          b(0)
        {}
    simple(int _a, int _b)
        : a(_a),
          b(_b)
        {}
public:
    int a, b;
};
```

And here is our Python script (with error checking removed to make it simple):

```
import kirpi
import simple

# Load config file with default behaviour
kirpi.Graph.loadConfigFile('')

# Load the graph and find its compile and link options
graph_file = 'python.graph'
graph = kirpi.Graph.load(graph_file)
(inc, lib, other) = graph.findCompilerOptions()
compiler_options = inc + other

# Load required shared libraries
(ok, static_libraries) = graph.loadLibraries(lib)

# Compile graph
graph.compile(False, compiler_options, static_libraries)

# Set graph input
a = simple.simple(2,1)
graph.setInput('input', a)

# Execute graph
graph.execute(False, compiler_options, static_libraries)

# Get graph outputs
out = graph.getOutput('out')
out2 = graph.getOutput('out2')
print('Result = ', '(', out.a, ',', out.b, ')', ',', out2)
```

Note the similarity to the *kpexec* example.

The *kirpi* Python module corresponds to the libkirpi Python binding, and the *simple* module corresponds to the `simple` class.

Boost.Python

We use pybind11 to create the Python binding of libkirpi, and all the graph's inputs and outputs are of type `kpPythonObject`. So if you pass the graph a Python object bound with Boost.Python, the graph will not recognize it.

If you want the graph to work with a Boost.Python object instead, the graph must be adapted as follows:

- it must retrieve a pointer to the input's underlying `PyObject` by calling `kpPythonObject::ptr()`
- it must convert the pointer to the desired Boost.Python class using Boost.Python's `extract` class template

Building and Running the Example

To build the Kirpi Python binding and the `simple` class implementation, run the `build.sh` script found in the `examples/python` directory.

To run the Python script (`test_python.py`) that executes the Kirpi graph (`python.graph`), run the `run.sh` script in the same directory.

6 FAQ

What platforms does Kirpi run on?

Kirpi runs on Linux (CentOS 7) and on Windows (Windows 7 or later).

Can my graph use the standard C++ library?

Yes. Just include the necessary header files, either in the nodes that need them or for the entire graph. Nodes need not be linked explicitly with the standard C++ library, because it is linked into *libkirpi*.

On Linux, *libkirpi* is linked with the standard C++ library that ships with gcc version 6.3.1. This makes it possible to execute Kirpi graphs on systems where older versions of the standard C++ library are installed. Graphs must be compiled against gcc 6.3.1's standard C++ library, however, as must any shared libraries they link with.

On Windows, *libkirpi* is linked with the standard C++ library provided by Microsoft. The library is compatible with versions 2015, 2017 and 2019 of Visual C++, and is installed with Kirpi if necessary. To compile graphs that use the standard C++ library, however, you must install Visual Studio for C++, which provides the necessary header files.

In *kpedit* can I reload or unload a Linux DSO used by my graph?

No. This is a limitation of Clang. You must restart *kpedit*. (On Windows, there is no such limitation, because JIT compilation is not currently supported.)

When I compile my graph, Kirpi says it cannot determine the type of an input or an output.

Because its type is not specified explicitly, and either it is not connected to anything, or it is connected to a node whose function is a template and whose parameter types cannot be determined by Kirpi.

In *kpedit* you can specify the types of a graph's inputs and outputs explicitly using the Graph Node Editor. Double-click on the *_inputs_* or *_outputs_* node, or right-click on it and select *Edit Node...* from the popup menu, then enter the types in the *Type* column.

When I try to set the value of a graph input, I get a "variable not found" message.

Ensure first that the graph has been compiled correctly. The graph's input and output variables are created in memory when the graph is compiled.

Second, it is possible that the variable is hidden and therefore not visible outside of the compiled module. This can happen when code is compiled with the `-fvisibility` option, or if you use a library that sets the option for you automatically (pybind11, for example).

Why does nothing happen when I execute my graph? Why is node "X" not executed?

Only nodes that are connected directly or indirectly to the graph's outputs are executed when the graph is executed. If a given node's outputs are not needed to compute the outputs of the graph, it is

not executed. And if nothing is connected to the graph's outputs, executing the graph does nothing. For more details about what happens when a graph is executed, see Graph Basics.