**Разработка абстрактной модели графа зависимости, её визуализация и применение**

**Содержание**

1. Структура ноды. Порты. Локальность данных. Механизм «dirty propagation» для оптимального пересчета графа

2. Порядок пересчета

3. Ребра. Правила подключений

4. События графа

5. Визуализация

6. Применение

7. Использованные ресурсы и литература

**Предисловие**

Всем привет, в этой статье я хочу рассказать о разработке нодового виджета на PySide, о том с какими проблемами мне пришлось столкнуться, и как я их решал.

Что такое граф мы можем узнать из математической теории графов, это раздел дискретной математики основным объектом изучения которой и являются графы. Граф - это совокупность вершин соединенных ребрами. Графы могут быть ориентированными, неориентированными, смешанными и изоморфными. Что касается терминологии, сразу хочу сказать, что у теории графов она еще не устоялась, и в разных публикациях под одними и теми же терминами могут пониматься разные вещи. В нашем случае мы рассматриваем направленный ациклический граф (Directed acyclic graph). Это такой граф, в котором отсутствуют направленные циклы, то есть пути, начинающиеся и кончающиеся в одной и той же вершине.

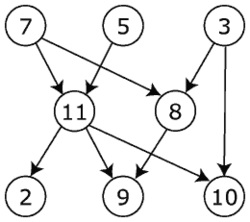


Рис.1 Направленный ациклический граф

Графы, или «нодовые редакторы» сейчас очень популярны и широко используются. Например, графы можно встретить в пакетах 3d графики или игровых движках, их применяют в науке, экономике, статистике, логистике, программировании и других дисциплинах. Понять, как это устроено, для меня было большим соблазном.

**Разработка**

**1. Структура. Локальность данных**

Граф – это объект, который хранит в себе ноды и связи между ними, определяет порядок пересчета блоков.

class AGraph(object):

def \_\_init\_\_(self, name):

super(AGraph, self).\_\_init\_\_()

self.object\_type = AGObjectTypes.tGraph

self.\_debug = False

self.\_multithreaded = False

self.name = name

self.nodes = []

self.edges = []

Нода – это основная единица графа, у нее есть входные и выходные атрибуты (порты), которые хранят в себе данные разных типов. Нода берет данные с входных портов, выполняет какое-то действие над ними, и записывает измененные данные в выходные порты.

class AGNode(object):

def \_\_init\_\_(self, name, graph):

super(AGNode, self).\_\_init\_\_()

self.graph = graph

self.name = name

self.object\_type = AGObjectTypes.tNode

self.inputs = []

self.outputs = []

class AGPort(object):

def \_\_init\_\_(self, name, parent, data\_type):

super(AGPort, self).\_\_init\_\_()

self.name = name

self.parent = parent

self.object\_type = AGObjectTypes.tPort

self.data\_type = data\_type

self.allowed\_data\_types = [data\_type]

self.affects = []

self.affected\_by = []

self.edge\_list = []

self.type = None

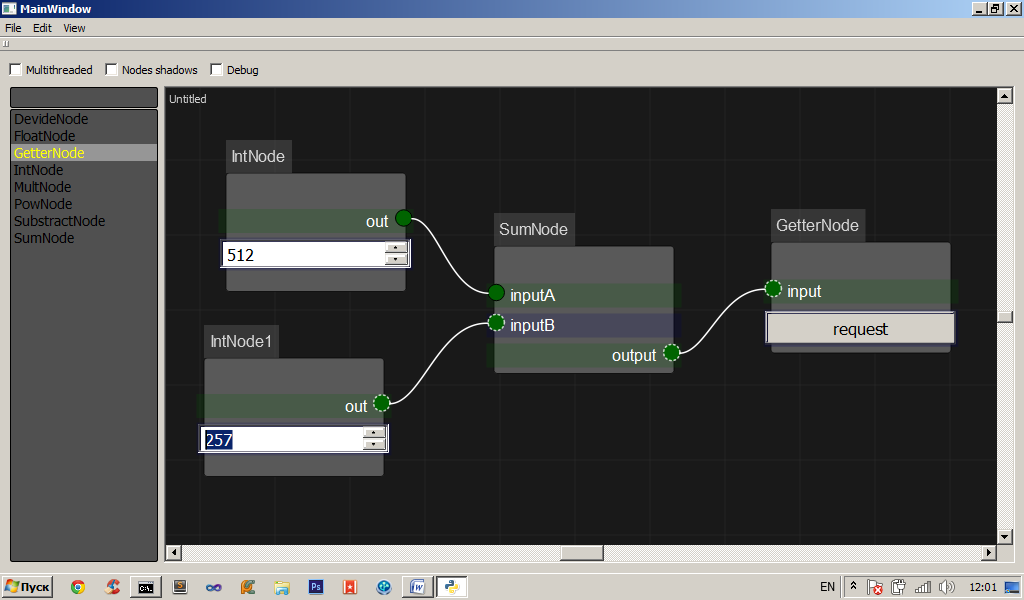
self.dirty = True

self.\_data = None

Порты могут соединяться с другими портами, образуя связи (Ребра). Ребро – это пара связанных нод.

Порты делятся по типам данных. У порта есть два списка. В первом списке хранятся ссылки на порты, на которые влияет данный порт, во втором – ссылки портов который влияют на данный порт. Нода не знает ничего из окружающей среды и никогда не использует данные, которые являются для нее внешними. Все что она знает это свои входные и выходные порты. Это одно из фундаментальных правил при проектировании нод. Подобная локальность данных позволяет легко добавлять новые ноды в граф и строить системы любой сложности.

Для оптимального пересчета графа я применил механизм «Dirty propagation», это запатентованная технология Autodesk. У каждого порта есть бинарное свойство «dirty». Например, если исходные данные изменились у какой-то одной ноды, то для получения результата на другом конце, нет необходимости пересчитывать все дерево нод целиком, достаточно посчитать только те ноды выходные данные которых потеряли актуальность. В момент изменения данных, вверх по графу расставляются флаги «dirty», обозначая маршрут для пересчета. Здесь я использовал рекурсивную функцию «push». Конечно же, реализация Autodesk и моя, могут различаться.



*Рис 4. Порты, обведенные белым пунктиром «грязные» и их ноды требуют пересчета.*

*Во время следующего запроса IntNode пересчитываться не будет, а SumNode возьмет данные из inputA.*

**2. Порядок пересчета**

Каждая нода вычисляется единожды, по глубине начиная с конца графа, учитывая «dirty» флаги. Ноды одного уровня не зависят друг от друга и могут вычисляться параллельно или последовательно на выбор. Ноды следующего уровня не начинают вычисляться до тех пор, пока не закончится просчет всех нод с предыдущего уровня, и так до того места от куда поступил запрос. Для этого я использовал метод графа «get\_evaluation\_odrer(node)» которая возвращает дневник вида

{

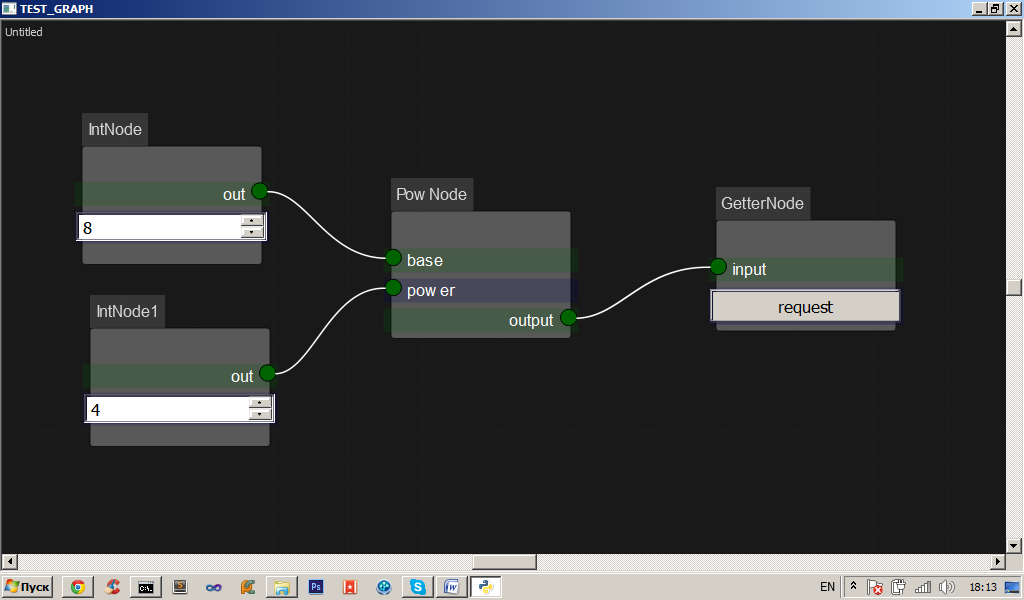
0: [node, node1,...],

1: [node3,...],

...

}

, где ключи - это номера слоев, а значения – списки нод которые нужно считать. В режиме «multithreaded» каждая нода слоя считается параллельно с остальными нодами.



2

1

*Рис 5. Порядок пересчета нод*

Ноды с первого слоя (IntNode, IntNode1) считаются и записывают данные во входные порты нод второго слоя (PowNode), с которыми соединены, затем ноды второго слоя считаются и записывают значения во входные порты ноды следующего слоя и так далее.

**3. Ребра. Правила подключений**

Ребро - это объект, соединяющий два порта, определяющий направление передачи данных (source -> destination). В абстракции такого класса нет, потому что, по сути, ребро – это два порта, списки влияния которых содержат ссылки друг на друга, за счет этого можно определить и направление передачи данных.

- Подключены, могут быть только входной и выходной порты с совместимыми типами данных.

- Входной порт может принимать только одну связь.

- Выходной порт может связываться со многими.

- Входной порт не может принимать значение с выходного порта собственной ноды.

Метод графа «add\_edge(source, destination)» проверяет все эти условия и создает связь, возвращая True, либо если условия не выполнены, ничего не делает и возвращает False.

**4. События графа**

1) Выходной порт ноды "А" соединился с входным портом ноды "Б".

В этот момент заполняются списки влияния портов.

Создается связь, и граф запоминает ссылку на нее.

Данные из выходного порта ноды "А" записываются во входной порт ноды "Б".

Расставляются dirty флаги.

Выполняется пустой метод, который можно перезаписать.

2) Связь между портами разорвалась.

Редактируются списки влияния.

Граф удаляет ссылку связь.

Выполняется пустой метод, который можно перезаписать.

3) Вызывается метод порта «set\_data(data)».

В этот момент по спискам влияния, выставляются флаги «dirty» вверх до конца графа, начиная с порта в котором изменились данные.

4) Порт запрашивает данные.

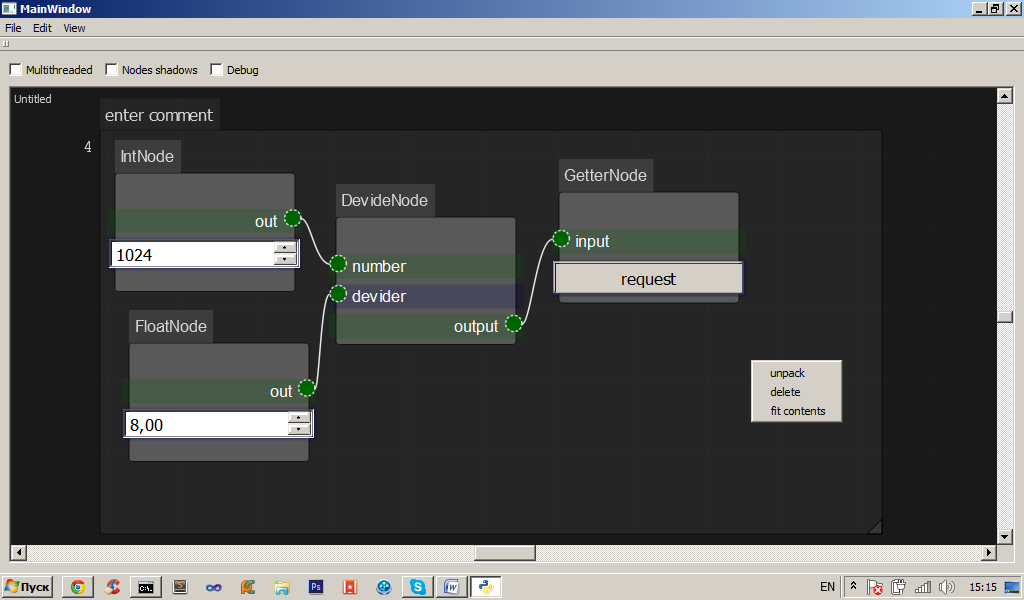
Проверяется dirty флаг, если True, то у графа запрашивается порядок пересчета нод, и они пересчитываются. Если False, берутся данные с последнего просчета.

**5. Визуализация**

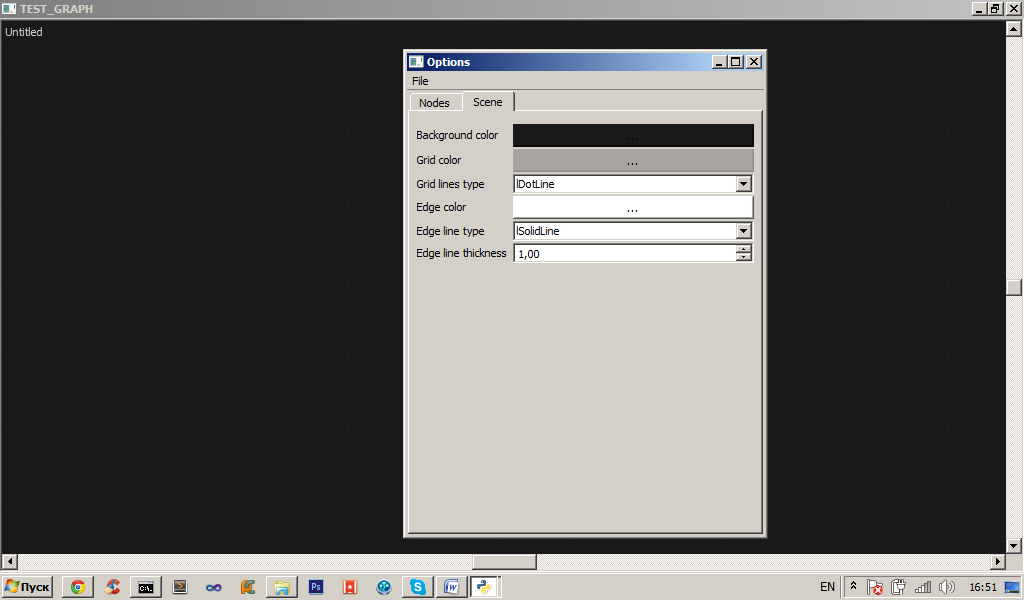
Для визуализации я выбрал Qt, так как этот Фреймворк пересекается с моей работой, да и лучшей альтернативы я просто не знаю.

Итак, на данный момент мы имеем работающий в консольке граф зависимости. Все что нам осталось делать - наследоваться от каждого из объектов графа, и рисовать, при необходимости дополняя методы. Графом будет класс, унаследованный от QGraphicsView и нашей абстракции виджета, нода – QgraphicsItem и абстрактная нода, ребро – QPainterPath или QGraphicsLineItem.

Есть объекты, которых нет в абстракции, например группировщик – объект, в который можно добавлять ноды, и подписывать комментарий. Этот объект наследуется от QGraphicsRectItem. Комментарий наследуется от QGraphicsTextItem, он многострочный. В момент переключения с него фокуса, геометрия автоматически выравнивается по левому правому углу ноды. Чтобы создать группировщик, нужно растянуть рамку с ctrl + shift. Добавить ноду в группировщик, можно перетащив её в пределы его геометрии, или в момент растягивания обеспечить пересечение геометрий ноды и будущего группировщика. Группировщик можно «распаковать» (извлечь из него все ноды), изменять его размеры при помощи специального элемента в правом нижнем углу, удалять. При удалении группировщика, его содержимое удаляется из графа. У группировщика есть функция автоматического изменения размеров под содержимое «fit\_contents()» которая работает на основе квадрата, который образуют ноды.

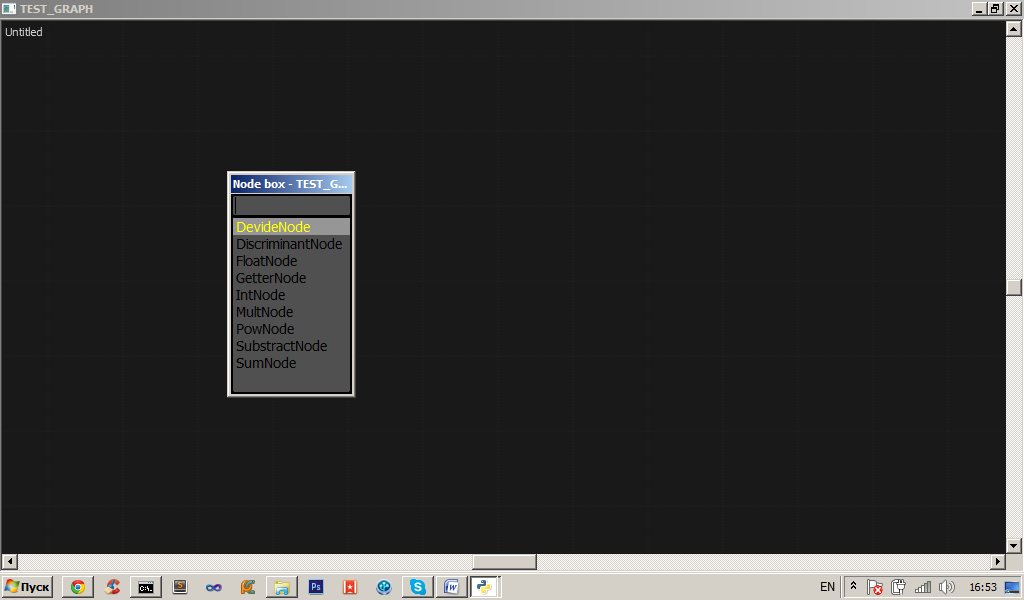


Виджет работает с файлом конфигурации QSettings. В контекстном меню есть пункт «Options», в котором можно настроить практически любую визуальную составляющую под себя. Изменения вступают в силу при перезагрузке приложения. Так же в конце сессии запоминаются положения окон, слайдеров и другие параметры.

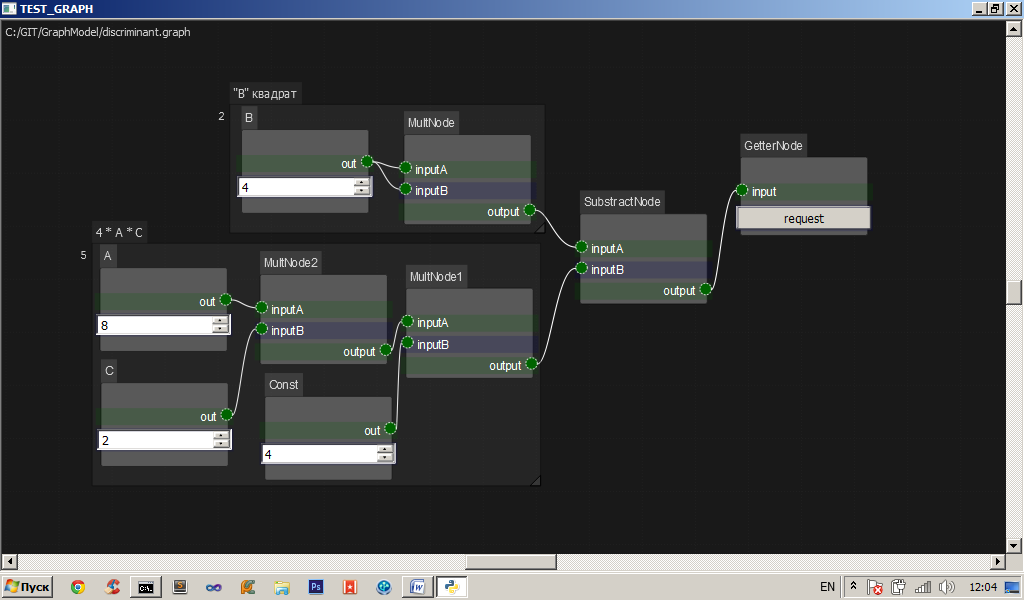


Сцену можно масштабировать и панорамировать. Работают горячие клавиши для операций сохранения, загрузки файлов. Формат файла представляет собой дневник, описывающий все данные графа.

У виджета есть «Node box». Это инструмент при помощи которого, динамически, создаются экземпляры нод. Вызывается он по нажатию кнопки «Tab». В верхней его части есть поле для поиска по имени. Граф, при добавлении ноды, выдает ей уникальное имя, проверяя имена существующих нод. Ноды можно выделять рамкой или с зажатым ctrl.

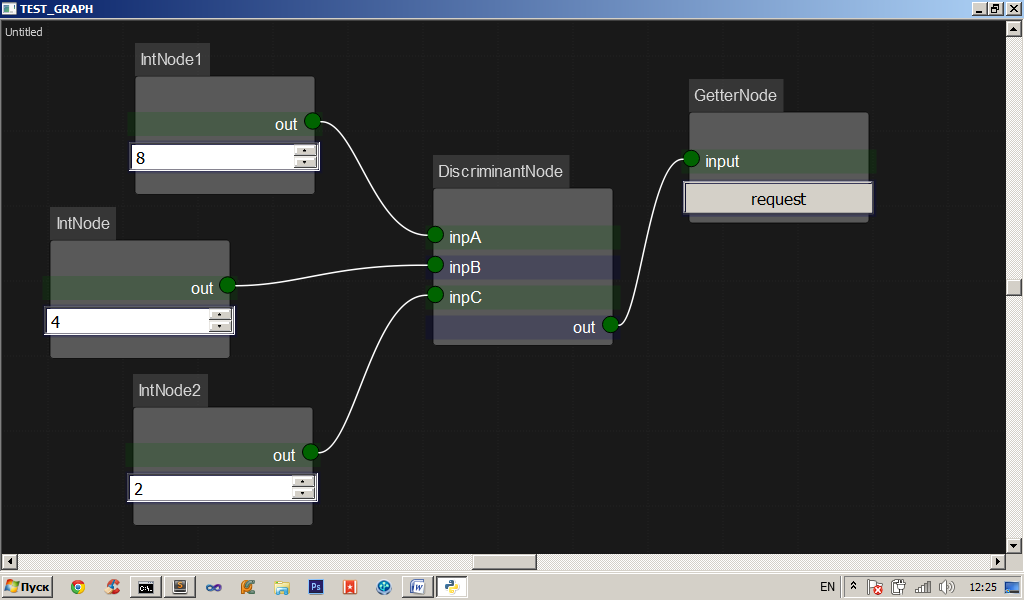


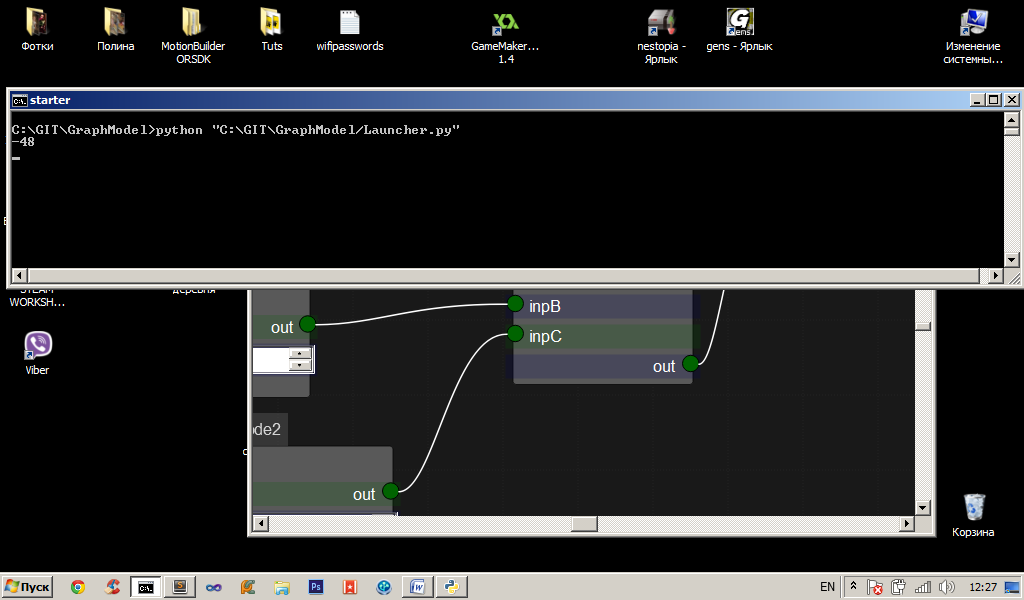
Давайте посчитаем дискриминант многочлена 8x2 + 4x + 2 = 0.



*Рис 6. Вычисление дискриминанта*

Конечно же, мы можем «завернуть» дискриминант в одну ноду.



**

*Рис 7, 8. Нода дискриминант*

**6. Применение. Перспективы**

Применять данный виджет можно в 3d пакетах, где есть Python. Например, Maya или Motionbuilder, в последнем, например, вообще нет нодового редактора. Добавляя собственные ноды, можно «заточить» виджет под любые нужды, например это может быть маппер персонажей и акторов в Motionbuilder или визуальная система сборки particle эффектов. Отдельным приложением его можно использовать как продвинутый калькулятор, или UML. Можно написать viewport для отображения графиков или объектов. Привязать генерацию кода на события соединения/разъединения портов. Из подобных разработок с открытым исходным кодом есть интересный проект «Coral» (<https://code.google.com/p/coral-repo/>), который уже давно не поддерживается и библиотека «PyQtGraph» Flowchart widget (<http://www.pyqtgraph.org/>).

**7. Использованные ресурсы и литература**

**1.** [**Wikipedia**](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D0%BB%D0%B0%D0%B2%D0%BD%D0%B0%D1%8F_%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%86%D0%B0)

# 2. Дэвид А.Д. Гоулд - Maya Полное руководство по программированию. Подробное описание языка MEL и интерфейса C++ API

# Отдельное спасибо хочу сказать Анатолию Юданову ([LinkedIn](https://ru.linkedin.com/pub/anatoliy-yudanov/6b/944/41a)), который давал крайне полезные советы в проектировании алгоритмов и общей архитектуры.