Правительство Российской Федерации

Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
Национальный Исследовательский Университет
«Высшая Школа Экономики»

Факультет Компьютерных Наук

Система для математической визуализации в реальном времени

Студент группы 401ПМ
Владимир Юрьевич Клепов

Научный руководитель:
доцент, кандидат наук
Алексей Антонович Никитин

Москва, 2015

Содержание

1 Введение

Изображения в математике заполняют пространство между символизмом и воображением. Гильберт в [12] отмечает, что математика разрывается между двумя тенденциями: стремлению к логически непротиворечивой символьной абстракции и к интуитивному пониманию задачи. Программа Декарта по алгебраизации геометрии сделала математику символьной наукой. Изображение стало инструментом догадок и примеров, но не корректного доказательства. В конце XIX века исследования абстрактных пространств ещё сильнее отдалили математику от интуиции. Недостаточно формальные утверждения часто не воспринимают как настоящую математику.

Символьная форма математических объектов точна, но далека от реальности. Учёному легко потерять связь с реальной задачей, которую он решает, и прийти к абсурдному результату. Сохранять эту связь — важное умение прикладного математика. Даже в чистой математике решение абстрактных задач начинается с анализа частных случаев, которые легко представить [17, 8]. Визуализация помогает учёным в каждом из этих случаев. Изображения делают науку проще и понятнее. Человек, не знакомый с формальной нотацией, легко увидит в изображении реальный объект.

Компьютер подходит к проблеме объединения символьной и визуальной математики сразу с двух сторон. Системы символьной алгебры и автоматического доказательства теорем решают формализованные задачи быстро, правильно и без участия человека. Неразрешимые в замкнутой форме задачи поиска корней, производных и интегралов приближённо решаются численными методами. Человеку остаётся придумывать идеи, корректно формализовывать прикладные задачи и следить за правдоподобностью результата. Математическое образование, построенное на формальных манипуляциях, не поможет в таких случаях. Современному математическому образованию нужна визуализация.

Компьютеры легко визуализируют сложные объекты. Современные учёные обрабатывают и отображают миллионы наблюдений, исследуют связь между ними и публикуют результаты в понятной форме. Это было немыслимо ещё несколько десятков лет назад. Математики больше не ограничены грубыми двумерными иллюстрациями на доске или бумаге. В [23] новая волна интереса к графике в математике называется «ренессансом визуализации». Другое преимущество компьютерной графики — создание интерактивных изображений. Меняя параметры системы и следя за изменением изображения пользователь лучше понимает модель.

Многие идеи для математической визуализации можно позаимствовать из смежных областей. Информационная графика — популярное и развитое направление визуализации. Практические рекомендации этой области собраны во множестве книг и статей (например, серия Тафти [20]). Теория автоматической визуализации не так обширна, но также развивается. Программисты и психологи исследуют взаимодействие человека с интерактивной графикой [5, 13].

1.1 Какой должна быть система математической визуализации

Хорошая система математической визуализации сузит конфликт логики и интуиции. Она находится на стыке разных дисциплин: формальных систем, численных методов, программирования и визуализации данных. Люди редко знакомы со всеми этими областями. Пользователь должен оперировать объектами на высоком уровне, а не наборами данных.

Система визуализации должна пользоваться всеми преимуществами своего носителя, компьютера. Пользователь хочет свободно исследовать математические объекты. Для этого взаимодействие с системой должно происходить в реальном времени. Другое эффективное использование компьютера — создание трёхмерной графики. Таким образом компьютерная система добавляет к бумажной графике два полноценных измерения: одно временное и одно пространственное. Человечеству понадобится ещё много времени, чтобы научиться извлекать пользу из такой свободы.

Использование системы для поддержки обучения добавляет ещё несколько требований. Образование должно быть свободным и общедоступным. Наша система должна быть кроссплатформенной, доступной даже на старых и мобильных устройствах. Стоит обдумать возможность демонстрации визуализаций на проекторе и даже печать черно-белых копий. Для этого системе нужны контрастные цветовые схемы.

Краткий список требований и уместных решений для системы:

- *простота:* высокоуровневый интерфейс и разумные умолчания.
- *доступность:* система на JavaScript доступна из браузера.
- *трёхмерность:* технология WebGL. Совместимость со старыми браузерами — через canvas или svg.
- *интерактивность:* эффективные модели вычислений и данных.

1.2 Обзор существующих систем

Компьютерные системы для работы с математикой поддерживают визуализацию. Maple, Matlab, Mathematica — мощные и универсальные пакеты. Но они не слишком полезны для визуализации: даже специальные образовательные версии стоят дорого, поддерживают только основные ОС, а для работы нужно изучать специальный язык программирования. Бесплатные пакеты — R, Octave, Julia решают только первую проблему.

Математические библиотеки для существующих языков — более прагматичный вариант. Пакеты SciPy и pyplot позволяют достаточно просто и бесплатно создавать визуализации на популярном языке python. Это гораздо лучше, но проблема совместимости остаётся: для взаимодействия с визуализацией нужен интерпретатор python, то есть мобильные устройства не смогут полноценно работать с такой системой.

Можно распространять визуализацию в виде изображения или видео. Но такая визуализация не интерактивна и никак не использует преимущества компьютерной среды. Для создания изображения в любом случае нужна математическая система.

Библиотека для визуализации данных d3.js — четвёртый по популярности проект на GitHub, который используют миллионы людей каждый день. Библиотека написана на популярном языке JavaScript и позволяет интернет-пользователям взаимодействовать с данными и находить в них интересные закономерности. Это уже близко к нашей цели, но остаются недостатки. Система не поддерживает трёхмерную графику. Преобразование математических объектов в набор данных — отдельная сложная задача, которую d3 не решает.

Специально для работы с математикой созданы онлайн-системы. Desmos не поддерживает трёхмерную графику. GraphyCalc работает только с явными функциями двух переменных. Wolfram Alpha производит вычисления на сервере, так что взаимодействие с изображением в реальном времени очень ограничено. Библиотека MathBox настолько сложна, что ей может пользоваться только её автор.

Доступные варианты не удовлетворяют требованиям к системе математической визуализации. Поэтому я привожу набор методов для построения системы. На этих методах основан grafar, быстрая JavaScript-библиотека для многомерной математической визуализации.

1.3 Структура работы

Глава 2 посвящена формальным моделям визуализации. Я рассматриваю классические работы Берте, Маккинли и Батлера. Далее я привожу удобную и гибкую декомпозицию процесса визуализации. На результатах главы основан общий подход к задаче и программная архитектура системы.

Глава 3 содержит концептуальную модель изображаемых объектов. Система ограничений — понятный и простой способ задать математический объект. Такие системы можно рассматривать как логические или геометрические объекты. Понятие зависимостей между переменными и прямого произведения систем помогут построить эффективные алгоритмы для работы с такими объектами.

Реактивное программирование — основная модель вычислений системы. В гл. 4 я привожу подробное описание парадигмы. Реактивные программы декларативно описывают зависимости между частями системы и прекрасно работают в реальном времени. Они концептуально похожи на системы ограничений.

Классические реактивные системы работают с атомарными переменными. Глава 5 содержит несколько обобщений парадигмы для работы с системами ограничений. Используя структуру зависимостей между переменными, можно построить эффективную модель данных для реактивных таблиц. Данные о близости точек можно хранить в виде графа. Моя система

поддерживает ограничения общего вида и может автоматически разрешать циклические зависимости.

В заключении (гл. 6) я ещё раз описываю основные результаты работы. Перспективные направления дальнейших исследований математической визуализации собраны в отдельной части.

Приложение А содержит примеры визуализаций, созданных в grafar. Глава начинается с простых двумерных визуализаций и завершается полноценными обучающими программами из проекта VisualMath.

2 Формальный подход к визуализации

Множество людей занимается визуализацией. Но накопленные ими знания разрознены, противоречивы и носят характер практических наблюдений. Такие правила работают лучше всего при создании бумажной визуализации. Автор полностью контролирует процесс: собирает и очищает данные, выбирает способ визуализации, пробует разные варианты, точно расставляет элементы и следит за печатью в типографии. При переходе к динамической визуализации всё меняется. Данные создаются в процессе визуализации и зависят от действий пользователя. Визуализации будут смотреть на разных устройствах: медленных, старых, с маленькими экраном. Автор уже не может сделать изображение идеальным в каждой ситуации — контроль больше похож на тестирование программ. Графический дизайн уже прошёл через похожий процесс. Чтобы система строила эффективные изображения, в неё нужно заложить формальное описание задачи визуализации.

Я не рассматриваю работы, анализирующие область с точки зрения философии математики [22]. Модели, использующие визуализацию как атомарный элемент в более полных теориях коммуникации [16] также исключены: польза изображений для математики обоснована во введении. Моя цель — построить систему, которая работает так, как ожидают пользователи. Я привожу несколько известных моделей визуализации, которые помогают взглянуть на эту область более формально. Далее я предлагаю многоступенчатую декомпозицию процесса визуализации. Результаты дают первое приближение к программной архитектуре системы.

2.1 Известные формальные модели

Жак Берте [1] одним из первых предложил формальную систему для классификации и оценки информационной графики в 1967 году. Автор использует семиотический анализ изображений: каждый элемент — знак, который обозначает (кодирует) какую-то информацию об исходных данных. Таким образом изображение — высказывание формального языка, состоящее из графических примитивов: точек, линий и областей. Значение каждого элемента определяют положение и различные визуальные переменные (retinal variables): форма, оттенок, яркость, размер, направление и текстура (рис. 2.1). Берте анализирует связь этих переменных со шкалами, в которых измеряются данные: номинальной, порядковой и абсолютной. Например, оттенок подходит для группировки объектов (номинальная шкала или отношение эквивалентности), но не позволяет эффективно сравнивать численные характеристики (абсолютная шкала). Работа Берте оказала огромное влияние на практику визуализации, предоставив удобную классификацию параметров изображения. Модель повлияла и на GL, язык программирования графических процессоров. Идеи Берте оказались очень гибкими, и на них основаны многие формальные модели визуализации. Хотя автор рассматривал только печатные материалы, аналогичные методы можно применить и для динамической компьютерной графики. Так, [18] добавляет к

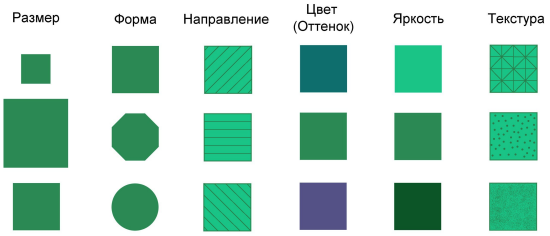


Рис. 2.1: Визуальные переменные Берте

визуальным переменным объём (для трёхмерных объектов) и прозрачность. Также можно рассматривать изменение изображения со временем.

В работе Джока Маккинли [14] предложен более формальный вариант модели Берте для построения автоматической системы визуализации. На основе примитивов Берте система создаёт разные варианты визуализации, сравнивает их и выбирает лучший. Маккинли действует по аналогии с теорией формальных языков. Для сравнения вводятся критерии "выразительности"и "эффективности". Выразительность — свойство изображения закодировать все факты о наборе данных и только эти факты. На основе исследований Кливленда [3] примитивы ранжированы по точности, с которой люди воспринимают их. На основе этих рангов вычисляется "эффективность"визуализации. Автор использует методы искусственного интеллекта, чтобы найти оптимальное изображение. Анализ в первую очередь направлен на визуализацию бизнес-процессов. Хиббард [11] более формально обосновывает модель в терминах теории решёток.

Последняя работа в моём обзоре представляет другую традицию формального анализа визуализаций. Батлер [2] описывает систему для научной визуализации и моделирования с упором на модель данных. Главная идея системы — представить научные данные в виде тривиальных расслоений. Этот топологический объект представляет собой прямое произведение двух пространств — базового пространства и слоя. Неформально говоря, к каждой точке базового пространства приделана копия слоя. Формализм обобщает функции одной или нескольких переменных. Базовое пространство соответствует свободным переменным, слой — зависимым. Конкретное отображение из базового пространства в слой называют сечением расслоения. Процесс визуализации состоит в применении отображений к объектам. Изображение становится набором графиков исходных пространств (в простейшем случае — координатных осей) и нескольких сечений. Кроме того, авторы предлагают таксономию методов визуализации по размерностям базового пространства и слоя. Способы снижения размерности — суперпозиция и серализация, то есть одновременное и последовательное отображение нескольких графиков.

2.2 Обобщённый процесс визуализации

Многие работы по теории визуализации [9, 14] предлагают разбиение процесса визуализации на более простые шаги. Такая декомпозиция помогает выделить известные подзадачи и подсказывает естественную программную архитектуру системы. В многоступенчатую структуру визуализации легко добавить модель взаимодействия с пользователем. Викерс [21] идёт ещё дальше и формализует разбиение в терминах семиотики и теории категорий. Такой подход кажется излишне формальным, но навязывает некоторую структурную дисциплину и помогает думать об эквивалентных способах визуализации.

Начнём с краткого описания объектов и отображений, участвующих в обобщённом процессе математической визуализации:

O_s Символьная форма объекта

A_{sp} Трансляция в язык программирования

O_p Программа

A_{pd} Вычисление

O_d Набор данных

A_{di} Визуализация

O_i Изображение

Для каждого типа объектов можно задать изоморфизм, определяющий эквивалентные объекты. Такая постановка задачи содержит другие похожие процессы. Оставив только $A_{di} : O_d \to O_i$, можно получить задачу визуализации данных. Постановка без O_s и A_{sp} — визуализация результатов компьютерной симуляции.

В современной математике объекты принято задавать формулами. Символьное описание — естественный для пользователя способ сообщить системе, что он хочет изобразить. Существует бесконечно много способов задать один и тот же объект. У каждой области математики свой взгляд — один и тот же объект может быть решением оптимизационной задачи или дифференциального уравнения, равновесием игры или условным математическим ожиданием. Методы компьютерной алгебры и символьных вычислений достаточно развиты, чтобы уже на этой стадии упростить объект. Некоторые операции, которые могут очень существенно упростить дальнейшую работу системы (например, поиск обратной функции), практически применимы только на этом уровне. Создание универсального языка математического программирования — слишком сложная задача для одного человека. Но добавление даже простейшего парсера с возможностью расширения делает работу с системой гораздо проще.

Чтобы построить объект, компьютеру необходим алгоритм на какомлибо языке программирования. Для бесконечных математических объектов

имеет смысл говорить об алгоритме, строящем конечную аппроксимацию. Изоморфизм программ — различные реализации одного и того же алгоритма. Эта задача алгоритмически неразрешима, но неформально можно воспринимать её как повод для поиска наиболее эффективной реализации. Я использую парадигму реактивного программирования (гл. 4), которая упрощает трансляцию и допускает эффективное взаимодействие в реальном времени. Пользователь будет часто взаимодействовать с системой именно на этом уровне.

Для визуализации нужен конечный набор данных. Существует множество наборов данных, удачно аппроксимирующих бесконечный математический объект. Чтобы выбрать оптимальное представление, необходимо учитывать множество факторов. Разрешающая способность изображения ограничивает необходимую точность: так (крайний случай), для экрана размером 1×1 пиксель достаточно показать, пусто ли отображаемое множество. Система должна работать в реальном времени, что также ограничивает число точек в наборе: слишком точную аппроксимацию невозможно эффективно обновлять. Выделение и формализация существенных фактов об отображаемом объекте — интересная тема для исследования, которую я обхожу в своей работе.

Изображение объекта — наша цель. Поскольку все средства отображения имеют конечные размер и разрешение, его можно рассматривать как матрицу над цветовым пространством [11]. Уровень такого описания слишком низок, чтобы работать с ним непосредственно. Более практичный способ — рассматривать изображение как множество графических примитивов Берте, и это совпадает с описанием объекта в языке GL. Но и такой уровень часто слишком низок для пользователя системы. Самый удобный высокоуровневый способ создать визуализацию — просто связать динамический набор данных с контейнером-изображением.

2.3 Выводы и перспективы

Изображение можно рассматривать как слово формального языка. Такой подход позволяет рассматривать визуализацию как преобразование формальных грамматик. Основные визуальные примитивы — точки, отрезки и треугольники. Присваивая им дополнительные визуальные параметры (например, цвет), можно закодировать дополнительную информацию. Эффективность кодирования зависит от свойств исходных данных. Чтобы совместить вычисления и визуализацию, необходима специальная модель данных. Для такой формальной предметной области, как математика, можно учитывать дополнительные параметры пространств, в которых лежат переменные.

Архитектуру системы можно естественно построить на описании черырёхступенчатой процедуры визуализации:

$$\text{формула} \to \text{программа} \to \text{данные} \to \text{изображение}$$

Каждый объект полезен для эффективной системы. Основная часть работы организована в соответствии с этой декомпозицией.

3 Формальное описание объекта

Я описываю системы ограничений — формальные объекты, подходящие для использования в системе математической визуализации. Попутно я обсуждаю соответствие между алгебраической записью, определяющей объект, и его геометрической формой. Обсуждение преследует двойную цель. Концептуальная модель и классификация ограничений уточняют необходимую программную архитектуру. Кроме того, базовая система символьной алгебры — необходимая часть будущих систем математической визуализации.

3.1 Системы ограничений

Я рассматриваю нелинейные системы уравнений и неравенств в действительных числах. Впрочем, все рассуждения этой части можно обобщить на переменные произвольных типов. Будем называть уравнение или неравенство $f(v_1, \ldots, v_n) = (\leq, \geq) g(w_1, \ldots, w_m)$ ограничением на (множество переменных) $V = \{v_1, \ldots, v_n\} \cup \{w_1, \ldots, w_m\}$. В дальнейших рассуждениях я для простоты игнорирую порядок параметров функций. Неоднозначность разрешается хранением данные в кортежах (см. 5.1). Более формальное обсуждение похожих идей приводит к комбинаторной логике и формальным моделям вычислений [4].

Ограничение на n переменных можно рассматривать как утверждение о точке пространства $\mathbb{R}^n$. Каждое ограничение C задаёт множество точек $S(C) = \{x \in \mathbb{R}^n | C(x_1, \ldots, x_n)\}$. Имеет смысл добавить к возможным видам ограничений явное перечисление, как в $x \in \{x_1, \ldots, x_m\}$, где x_i — числовой литерал. Примеры ограничений на $\{x, y\}$: $x + y \leq 0$, $x = 2y$.

Одного ограничения недостаточно. Чтобы работать с системами ограничений, нужны правила комбинирования. Пусть $P(v_1, \ldots, v_n)$ и $Q(w_1, \ldots, w_m)$ — ограничения, причём $\{v_1, \ldots, v_n\} \cap \{w_1, \ldots, w_m\} = \emptyset$. Утверждения в таком случае выполняются независимо друг от друга, и $S(P \wedge Q) = S(P) \times S(Q)$. Поясним на примере: $S(0 \leq x \leq 1 \wedge 0 \leq y \leq 1) = S(0 \leq x \leq 1) \times S(0 \leq y \leq 1) = [0, 1] \times [0, 1]$. Назовём такой способ снижения размерности задачи разложением в прямое произведение.

Другой интересный случай — функциональная зависимость между переменными. Пусть ограничение $M(p_1, \ldots, p_n, x)$ имеет вид $x = f(p_1, \ldots, p_n)$. Рассмотрим такое ограничение в контексте системы $C_1 \wedge \cdots \wedge C_m$. Пусть $P_1, \ldots, P_m$ — все ограничения на переменные $p_1, \ldots, p_n$. Тогда справедливо утверждение $S(P_1 \wedge \cdots \wedge P_m \wedge M) = \{(\overline{p}, f(p)) \, | p \in S(P_1 \wedge \cdots \wedge P_m)\}$. Такие ограничения играют очень важную роль в нашей модели вычислений, и мы будем называть их направленными.

Иногда ограничений в системе не хватает для выделения функциональной зависимости. Например, значение x в системе $y = \sin(x)$ не определено. Формально правильное, но совершенно бесполезное действие в такой ситуации — добавить к системе пустое ограничение, $x \in \mathbb{R}$. На практике нужно ограничить область значений x (подробнее в части 4.5) или запретить такие недоопределённые системы.

Ограничения других видов будем называть ненаправленными. Они действуют на все переменные, входящие в него. Чтобы понять идею, представим себе систему $x \in [-2, 2] \wedge y = 2x \wedge y \geq 1$. Первые два ограничения задают направленную зависимость y от x. Но на y действует также ненаправленное ограничение $y \geq 1$. Не каждое значение x задаёт корректный y, и второе ограничение начинает действовать в обратную сторону. Ситуация изображена на рис. **??**. Встретив ненаправленную зависимость, мы не можем применять наши правила разбора и переходим к численным методам решения систем ограничений.

Наконец, будем называть свободными переменными системы переменные, которые не зависят от других переменных. Так, x — свободная переменная в $x = 0$, $x \in \{1, 2, 3\}$, $x \leq 12 \wedge x \geq 0$, но не в $t = 0 \wedge x = \sin(t)$. Но по такому определению x также свободная переменная в $x^2 = 0$, ведь такая система эквивалентна $x = 0$. Это подсказывает обобщение. Будем называть $B = \{v_1, \ldots, v_n\}$ свободным набором переменных системы, если переменные в наборе зависят только друг от друга, и никакое подмножество B не является свободным набором. Например, в $x^2 + y^2 = 1$ свободный набор — $\{v_1, \ldots, v_n\}$. Если S — система ограничений, будем обозначать множество всех переменных, входящих в свободные наборы, $base(S)$. Все остальные переменные системы можно выразить через свободные, и они полностью определяют структуру описываемого объекта. Более подробно их влияние исследуется в 5.2, 5.3.

Приведу концепцию алгоритма, упрощающего систему ограничений. Алгоритм последовательно применяет правила разложения в прямое произведение и выделения направленных ограничений. Работа заканчивается, когда ни одно из этих правил не применимо, и значения оставшихся переменных ищутся численно. На примере:

$$
\begin{gathered}
S(z = ax + y \wedge y \in [0, 1] \wedge x^2 + a^2 = 1) = \\
\big\{(a, x, y, ax + y) | (a, x, y) \in S(y \in [0, 1] \wedge x^2 + a^2 = 1)\big\} = \\
\big\{(a, x, y, ax + y) | (a, x, y) \in S(x^2 + a^2 = 1) \times [0, 1]\big\} = \\
\big\{(a, x, y, ax + y) | (a, x, y) \in \big\{a, x | x^2 + a^2 = 1\big\} \times [0, 1]\big\}
\end{gathered}
$$

Таким образом вместо решения системы в 4 неизвестных, необходимо решить всего одно уравнения с двумя неизвестными. После того, как в гл. 4 мы познакомимся с парадигмой реактивного программирования, идею алгоритма можно будет реализовать более эффективно. Пока же отметим, что три основных вычислительных операции в системе — декартово произведение, отображение и решение системы ограничений.

3.2 Приведение ограничений

Из определения направленных ограничений видно, что их можно эффективно вычислять. Иногда более общие ограничения можно привести к такому виду символьными манипуляциями. Для любой обратимой функции

g ограничение $g(x) = f(\dots)$ эквивалентно $x = g^{-1}(f(\dots))$. Если функция $g(p_1, \dots, p_n, x)$ представима в виде $g_1(p_1, \dots, p_n) + g_2(x)$, где g_2 обратима, ограничение также можно направить в x. В некоторых случаях, например, $x = y$, существует несколько способов выбрать зависимую переменную. Чтобы сделать правильный выбор, необходим более подробный структурный анализ зависимостей, инструменты для которого мы получим в следующей главе. Введение в методы символьного решения систем зависимостей можно найти в [19].

Ограничения вида $v \in \{v_1, \dots, v_n\}$ в принципе позволяют выражать объединение конечного числа областей. Рассмотрим одномерный случай: выражение $x \in \bigcup_{i=1,\dots,n} [x_{i0}, x_{i1}]$. Эквивалентная система ограничений — $i \in \{1, \dots, n\} \wedge t \in [0, 1] \wedge x(i, t) = x_{i0} + \frac{t}{x_{i1} - x_{i0}}$. Аналогичные параметризации можно ввести и для несвязных областей высших размерностей, хотя это гораздо сложнее. В приложении А есть несколько примеров использования такого способа для построения графиков разрывных функций.

3.3 Дальнейшее развитие

Я рассматриваю только небольшой класс математических выражений, уравнения и нестрогие неравенства в действительных числах. Это вызвано двойной ролью символьных представлений в моей системе: концептуальной модели и удобного интерфейса. Думать о более общих системах ограничений полезно, даже система не будет обрабатывать их автоматически.

Множество появляется в модели только в виде синтаксического сахара, но не полноправного элемента. Это сделано намеренно: произвольные конструкции с множествами требуют типизации переменных и усложняют систему. Однако операции на множествах — мощный инструмент, который позволил бы расширить класс описываемых систем. Многие операторы: min, max, $Conv$ (выпуклая оболочка), среднее значение — естественным образом определены именно на множествах, а не на числовых переменных. Система поддерживает их через программный интерфейс.

Направленное ограничение $y(x, a) = x^2 + a$ можно рассматривать не как утверждение о зависимости между x, y, a, а как определение функции двух переменных, y. Эту функцию можно использовать как $z = y(2, 3)$ или даже $z(a) = y(3, a + 2)$. На таких функциях первого класса можно задать операторы дифференцирования и интегрирования. Реализация таких функций — очень сложная задача. Я более подробно остановлюсь на двух частных случаях, не требующих полноценной поддержки функций.

Частные производные: для переменных x, y часто достаточно легко определить оператор дифференцирования, $\frac{\partial y}{\partial x}$. Используя механизм списковых переменных со структурной информацией, который я предлагаю в следующей главе, значение такого выражения можно найти численно.

Распознавание и обработка задач Коши: $y' = f(x, y) \wedge y(x_0) = y_0$, x_0, y_0 — числовые литералы. Два таких уравнения эквивалентны ненаправленному ограничению $F(x, y) = 0$, но требуют специальных численных методов.

Математические модели физических процессов часто основаны на дифференциальных уравнениях, и поддержка такой записи была бы полезна.

3.4 Выводы

Математические объекты удобно записывать системами ограничений. Формальный логический анализ таких систем показывает два способа упрощения: разложение в прямое произведение и выделение функциональных зависимостей. В первую очередь такая модель — концептуальная основа системы визуализации. Но возможность задавать объекты в символьной форме также помогла бы пользователям.

4 Реактивное программирование

Многие системы ограничений удобно моделировать, анализируя явные зависимости между компонентами (гл. 3). Реактивное программирование (РП) — парадигма, основанная на такой же идее. РП хорошо проявило себя в двух классах задач: создании систем реального времени (программы на языке Lustre [10] управляют самолётами и атомными электростанциями) и создании пользовательских интерфейсов (библиотека bacon.js). В нашем случае РП хорошо моделирует предметную область и позволяет эффективно использовать численные методы.

Реактивное программирование — одна из реализаций декларативной парадигмы. В императивном программировании программист явно описывает последовательность действий, выполнение которых изменяет состояние программы и приводит к нужному результату. Декларативная программа описывает желаемые свойства результата, и по этому описанию язык сам строит алгоритм. К декларативным языкам часто относят также функциональное и логическое программирование. Популярные декларативные языки — SQL и CSS.

Реактивное программирование основано на программной архитектуре с потоками данных (data-flow programming). Программист задаёт ограничения на значения переменных, которые должны выполняться в любой момент времени. В случае направленных ограничений зависимости можно явно представить в виде графа потока данных. При изменении значения переменной система автоматически обновляет другие переменные, чтобы вернуться в корректное состояние, то есть реагирует на событие. Пример реактивной системы, с которой знаком каждый — редактор таблиц, например, Microsoft Excel. Пользователь связывает значение ячейки с другими ячейками формулой. На похожих принципах строят системы визуального программирования, например, пакет для процедурного моделирования Grasshopper. Система управления пакетами apt-get тоже работает с графами зависимостей.

В этой главе я привожу краткое описание самой парадигмы на основе [15, 6, 7] и своей реализации. Графы потока данных — важный инструмент структурного анализа зависимостей. В нашем случае pull-реактивная реализация с кешированием даёт лучшую производительность, хотя в некоторых случаях выгодно перейти на альтернативную реализацию. Реактивное программирование упрощает создание пользовательских интерфейсов. Специальные обобщения реактивной модели для работы с системами зависимостей приведены в следующей главе.

4.1 Основные понятия

В реактивной системе присутствуют объекты двух типов: реактивные переменные RV и отображения между ними, $RM : RV^n \to RV$. Реактивная переменная — полиморфный контейнер для значений, меняющихся со временем, и на $v : RV$ в момент t задан оператор $val_t(v)$, который возвращает

актуальное значение. В нашей реализации процесс стационарен, то есть не зависит от предыдущих значений переменных.

Для простоты предположим, что все реактивные переменные имеют тип действительных чисел. Для функции $f : \mathbb{R}^n \to \mathbb{R}$ и $n+1$ реактивной переменной $v_{1,\dots,n}, d : RV$ можно задать оператор поднятия (lifting). Оператор связывает переменные направленным ограничением: $lift\,(f, (v_1, \dots, v_n), d)$ так, что $\forall t \Rightarrow val_t(d) = f(val_t(v_1), \dots, val_t(v_n))$. Переменная d становится связанной. Чтобы найти значение d, нужно знать все значения $v_{1,\dots,n}$.

Реактивная система взаимодействует со своим окружением. К каждой реактивной переменной можно обратиться двумя способами: считать значение и записать новое. Эти взаимодействия называются, соответственно, pull и push. В классических реализациях push-обращение к связанной переменной оставляет систему в неопределённом состоянии: для произвольных значений $v_{1,\dots,n}$ соотношение $f(v_1, \dots, v_n) = c$ не выполнено. Сделав элементами интерфейса не переменные, а отображения [7], можно избежать таких проблем: push-обращение не записывает значение, а переопределяет функцию.

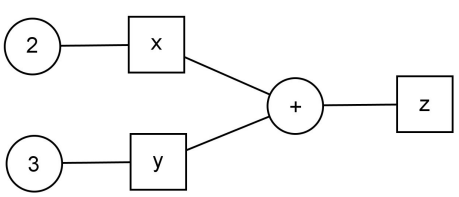
Основная метафора парадигмы — связь с микроэлектроникой. Отображения соответствуют компонентам, переменные-сигналы движутся по дорожкам на плате. Это приводит нас к понятию графа потока данных, общему для всех методов программирования с потоками данных.

4.2 Граф потока данных

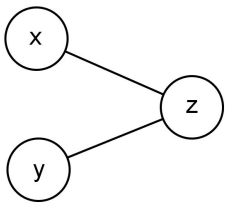
Граф потока данных — удобная интерпретация реактивной системы. Через такой граф гораздо проще анализировать связи между элементами.

Будем называть графом потока данных (в полной форме) двудольный граф $G = \langle V, E \rangle$. Множество вершин V состоит из элементов системы — реактивных переменных и отображений. Пусть C — множество реактивных переменных, M — отображений. Тогда $V = M \cup C, M \cap C = \emptyset$. Дуги связывают переменные с функциями, то есть $\forall (e_1, e_2) \in E \Rightarrow e_1 \in C \wedge e_2 \in M \vee e_1 \in M \wedge e_2 \in C$. Системе ограничений $x = 2 \wedge y = 3 \wedge z = x + y$ соответствует полный граф потока данных, изображённый на рис. 4.1а. Граф потока данных не содержит циклов, иначе возникает циклическая зависимость, и корректного порядка вычислений не существует.

Часто граф потока данных можно описать более простым способом. Заметим, что оператор $lift$ не позволяет создавать вершину $m \in M$, из которой не выходят дуги. Более того, если переменная $u \in C$ не входит ни в одно ограничение, её значение не определено и программа не выполнима. В корректном графе потока данных E — сюръекция $M \to C$. Если одну переменную $c \in C$ связывает несколько ограничений, $\exists c \in C : (m_1, c) \in E \wedge (m_2, c) \in E, m_1 \neq m_2$, система накладывает дополнительные ограничения на параметры функций m_1, m_2. Значит, такая структура приводит к недопустимому графу потока данных. Формально E задаёт инъекцию из $C \to M$. Значит, между реактивными переменными и функциями существует биекция, и можно рассматривать только один тип элементов.



(a) Полная форма



(b) Сокращённая форма

Рис. 4.1: Граф потока данных для системы $x = 2 \wedge y = 3 \wedge z = x + y$

Сокращённый граф потока данных, $G = \langle V, E \rangle$, определяет любую корректную реактивную систему. Вершины такого графа — пары $\langle m : RM, c : RV \rangle$. $fun(v)$ сопоставляет такой вершине числовую функцию. Можно интерпретировать граф как состоящий из функций и получить классическую структуру из формальной логики. Думать о вершинах как о величинах, зависимости между которыми выражены дугами — интуитивная, физическая интерпретация. Сокращённый граф для прошлого примера изображён на рис. 4.1b. Больше примеров — в приложении A.

4.3 Исполнение реактивных программ

Pull-обращение к вершине $v \in V$ требует вычисления актуальных значений всех $p : (p, v) \in E$. Это совпадает с обходом в глубину. Наивная реализация использует рекурсию:

```
NaivePull(v):
    for p: (p, v) ∈ E
        val_i = NaivePull(p)
    return fun(v)(p_1, ..., p_n)

NaivePush(v, fun)
    v.fun = fun
```

Использование такого алгоритма ведёт к экспоненциальной сложности на некоторых графах. Предположим, что вычисление значения каждой вершины занимает одинаковое время, 1. Время на обход графа пренебрежимо мало по сравнению с самим вычислением. Последнее предположение близко к истине, первое выполняется для графов с небольшим числом истоков (подробнее в следующей главе). Рассмотрим граф особого вида, изображённый на рис. 4.2 со сливом t. Общее число вершин $N = 2(n + 1)$. При $n = 0$ Pull(t) потребует $T_0 = 2$ времени. Заметим, что $T_i = 2T_{i-1} + 1 = 3 \cdot 2^i - 1$, то есть алгоритм имеет экспоненциальную сложность.

Вариант с кешированием результатов гораздо эффективнее. Этот метод и его варианты, мемоизация и табуляция — стандартные техники динамического программирования.

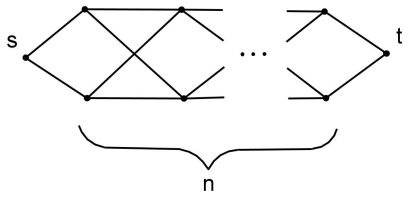


Рис. 4.2: Граф потока данных с экспоненциальной сложностью вычисления

```
Pull(v, isValid, memo):
    if v ∉isValid
        for p: (p, v) ∈ E
            p_i = Pull(p)
        memo[v] = fun(v)(p_1,...,p_n)
        isValid = isValid ∪ {v}
    return memo[v]

Invalidate(v, isValid):
    if v ∈ isValid
        isValid = isValid \ {v}
        for c: (v, c)∈ E
            Invalidate(c)

Push(v, fun):
    v.fun = fun
    Invalidate(v)
```

Как и в аналогичной реализации поиска в ширину, $T(Pull, v) = |Pred(v)|+1$, где $Pred(v)$ — множество всех вершин графа G, из которых достижима v. В худшем случае операция займёт $O(|V|)$. При этом результат не зависит от конфигурации графа. Такой алгоритм оптимален по времени: $T(v) \geq |Pred(v)+1|$, поскольку все значения, от которых зависит v, всё равно нужно вычислить. $T(Push, v) \approx 0$.

У динамического алгоритма есть недостаток: он использует больше памяти, $O(|V|)$ вместо $O(1)$ у наивного алгоритма. К тому же на некоторых графах (деревьях) алгоритмы работают за одинаковое время. В таком случае динамическая версия, скорее всего, будет работать медленнее за счёт лишних обращений к памяти.

4.4 Множественные запросы

Существует ещё одна динамическая реализация реактивной системы. Будем называть предыдущую push-реактивной, а новую — pull-реактивной. Push-реализация вычисляет значения переменных сразу же при изменении, то есть вычисления инициирутся через push. Сложность функций симметрична pull-реализации и составляет $T(AltPull) = O(1)$, $T(AltPush) = O(|V|)$.

(a) Обращения к реактивной системе (b) Задержка при pull-реактивности

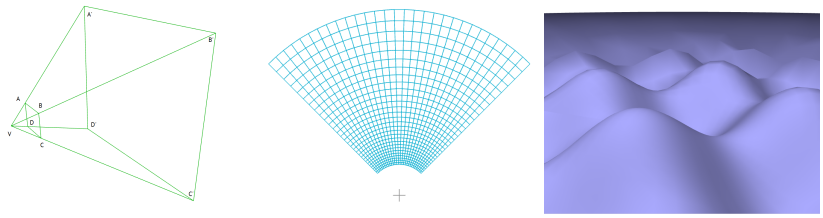
Рис. 4.3: Взаимодействие с реактивной системой

Чтобы сравнить эффективность push- и pull-реактивных реализаций, рассмотрим типичный паттерн взаимодействия с реактивной системой (рис. 4.3a). Устройство отображения обновляется с определённой частотой (обычно 60 кадров в секунду). Обновлять информацию чаще не имеет смысла. Pull-запросы поступают в систему через приблизительно равные промежутки времени, 60 раз в секунду. При этом в течение кадра пользователь может сделать несколько pull-обращений, потому что компьютер быстро реагирует на нажатие клавиш.

В pull-реактивной системе m идущих подряд Pull-запросов ко множеству вершин Q займут $T \approx |\bigcup_{v \in Q} Pred(v)| \leq O(|V|)$. В push-реактивной все значения уже вычислены перед pull-запросом, и любое их число выполняется моментально. Для m идущих push-запросов складывается противоположная ситуация. В pull-реализации push-запрос выполняется моментально. Но время на выполнение push-запросов в push-системе складывается, $T \approx \sum_{v \in Q} |Succ(v)| \leq m|V|$, где $Succ(v)$ — множество всех вершин, достижимых из v.

При нашем способе использования pull-реализация работает эффективнее. Часто все pull-запросы в течение кадра направлены на одну вершину: пользователь зажимает клавишу, управляющую значением параметра или двигает ползунок. Назовём целевую вершину v. Если произошло m изменений, pull-реализация отработает 1 раз (для последнего значения v), а push-реализация — m раз. Конечно, без предложения о стационарности процесса эта оптимизация перестанет работать, потому что все промежуточные значения v имеют значение.

Push-реализация лучше использует время внутри кадра. В pull-реализации между pull-запросами система простаивает, а все вычисления выполняются в начале кадра. Если вычисления занимают время t, новый кадр появится на t позже pull-запроса. Если ожидаемая частота запросов — m за кадр, а изображение обновляется каждые T, выгодно было бы выполнять push-запросы, вычисления для которых займут менее $\frac{T}{m}$, моментально. Впрочем, интерпретатор JavaScript обычно способен справиться с такой ситуацией самостоятельно, а задержка в несколько миллисекунд незаметна человеку. Если есть повод считать, что изменений больше не будет (например, пользователь отпустил кнопку мыши), pull-запрос можно подать вручную и использовать всё время до конца кадра.



(a) Область видимости (b) Адаптивная сетка (c) Результат

Рис. 4.4: Адаптивная детализация

4.5 Реактивные интерфейсы

Ещё одно преимущество реактивной модели — отличная совместимость с любыми интерфейсами. Используя механизм наблюдателей, который поддерживают все версии JavaScript, любой элементу страницы при изменении может вызвать функцию. Если функция отправит в реактивную систему push-запрос, изображение автоматически изменится.

Другой источник запросов — таймер. JavaScript позволяет вызывать функции через заданные промежутки времени, причём в большинстве современных браузеров можно синхронизировать частоту с частотой обновления экрана. Самое важное для нас — обновлять изображение, так что для каждого активного графика по таймеру делается pull-запрос. Разумеется, если данные не изменились, никаких вычислений не происходит. Также таймер можно использовать для создания анимации: на каждом кадре в систему передаётся push-запрос.

Движение камеры также может инициировать изменение параметров. Таким образом можно строить графики с адаптивным уровнем детализации. Если одна из осей графика — свободная переменная системы, из положения камеры можно извлечь информацию о видимом интервале этой переменной. Для двумерной визуализации это тривиально — достаточно хранить прямоугольник $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$. Более интересен трёхмерный случай. Конечно, можно так же извлекать параллелепипед, который содержит всю видимую область. Но область, видимая виртуальной камерой, точно описывается усечённой прямоугольной пирамидой (на рис. 4.4a камера — в точке V). Использование линейной перспективы означает, что чем дальше от камеры находятся точки, тем ближе друг к другу они кажутся. Уместно ввести нелинейное преобразование, которое распределит точки в видимой области так, что при проекции на изображение расстояние между ними будет примерно равным (рис. 4.4b, камера на крестике). Конечно, если на большом удалении изображаемая поверхность не является приближённо линейной, дальние участки изображения превращаются в визуальный мусор. Но информация о том, что функция странно ведёт себя вдалеке тоже может быть полезной. Пример визуализации с адаптивной детализацией — рис. 4.4c. На переднем плане поверхность более гладкая.

5 Реактивное программирование для систем ограничений

Обычно реактивные системы работают с атомарными переменными, но в нашем случае необходимо обрабатывать большие таблицы данных. Анализ зависимостей позволяет выбрать оптимальный способ и существенно ускорить работу в системах со множеством свободных переменных. Хотя РП непосредственно работает только с направленными зависимостями, я предлагаю способ реализации более общих ограничений, который также разрешает циклические зависимости. Информацию о структуре объекта удобно хранить в виде графа.

5.1 Кортежи

Для следующей работы будет полезно обобщить понятие вектора. Определим (неупорядоченный) кортеж как $t = \langle I, D, F \rangle$, где I — множество индексов, D — множество элементов, $F : I \to D$. Кортеж можно представить как множество пар $\langle i \in I, d \in D \rangle$. Вектор в $\mathbb{R}^n$, — частный случай кортежа с индексами $I = \{1 \dots n\}$.

На любом конечном множестве индексов можно ввести линейный порядок. Не ограничивая общности, положим $I \subset \mathbb{N}$. Любой кортеж t такого вида можно упорядочить по индексам: $odr(t) = (\langle i_1, d_1 \rangle, \dots, \langle i_n, d_n \rangle)$, $\forall a, b : 1 \leq a < b \leq n \Rightarrow i_a < i_b$. Для краткости можно опускать индексы.

На таких кортежах можно определить коммутативное декартово произведение. Пусть $t_1 = \langle I_1, D_1, F_1 \rangle, t_2 = \langle I_2, D_2, F_2 \rangle$, причём $I_{1,2} \subset \{1 \dots n\}, I_1 \cup I_2 = \emptyset$. Тогда $p = t_1 \times t_2 = t_2 \times t_1 = G : I_1 \cup I_2 \to D_1 \cup D_2$, где p — также упорядочиваемый кортеж.

Другой полезный оператор — проекция $a_J = proj(a, J), J \subseteq I(a)$. Для любого подмножества индексов проекция создаёт кортеж, содержащий некоторые элементы исходного кортежа.

5.2 Работа с переменными-списками

Интересные математические объекты и наборы данных почти всегда содержат более одной точки. Поэтому, в отличие от традиционных реактивных систем, наша система работает не с атомарными переменными, а с векторами. Физически такие объекты реализуют на основе массива. Далее я приведу формальное описание модели данных, которую я использую для хранения объектов. Основные требования к модели — возможность хранить структуру объекта и эффективное использование памяти. Длительное хранение данных не требуется (хотя полезно импортировать данные разных форматов), но нужно обеспечить быстрое взаимодействие с GL.

Одна из самых развитых технологий хранения структурированных данных — реляционные базы данных. Но эта модель разработана с другими приоритетами. В научных приложениях, где требуется быстрая обработка, данные часто хранят в n-мерных массивах (Matlab, NumPy, Fortran90,

HDF5). В такой модели появляется избыточность, к тому же реализации многомерной адресации на JavaScript неэффективны. Для системы визуализации нужен гибридный подход.

Таблица T — аппроксимация n-мерного объекта. Я хочу сохранить модель, в которой каждая реактивная вершина содержит только свои данные. Далее я использую упорядоченные кортежи, введённые в части 5.1. Определим таблицу как кортеж столбцов: $T = \langle\langle i_1, c_1\rangle, \ldots, \langle i_n, c_n\rangle\rangle$, где $\forall i\, c_i \in \mathbb{R}^m$. С таким определением оператор $I(T)$ — заголовок таблицы. Таблицу можно рассматривать как множество точек в $\mathbb{R}^n$. Напомню, что для нашей системы нужны две операции: декартово произведение таблиц и добавление столбца.

В нашей модели данные хранятся по столбцам, и нужно определить декартово произведение в таких терминах. Для этого введём конкатенацию двух векторов: $vw = (v_1, \ldots, v_n)(w_1, \ldots, w_m) = (v_1, \ldots, v_n, w_1, \ldots, w_m)$. Будем обозначать

$$v^k = \underbrace{vv\ldots v}_{k}$$

Далее, определим скалярное произведение (растяжение) как

$$k \cdot (v_1, \ldots, v_n) = (\underbrace{v_1, \ldots, v_1}_{k}, \underbrace{v_2, \ldots, v_2}_{k}, \ldots, \underbrace{v_n, \ldots, v_n}_{k})$$

Заметим, что $(n \cdot v)^k = n \cdot (v^k)$, $(v^m)^k = v^{mk}$, $n \cdot (k \cdot v) = (nk) \cdot v$. Такие операции можно поэлементно применять и к таблицам.

Начнём с таблиц, содержащих один столбец. Пусть $A = \langle\langle i_a, d_a\rangle\rangle, B = \langle\langle i_b, d_b\rangle\rangle$. Тогда

$$A \times_T B = B \times_T A = \begin{cases} A^{|B|} \times |A| \cdot B & i_a < i_b \\ |B| \cdot A \times B^{|A|} & i_a > i_b \end{cases}$$

Ясно, что такое произведение таблиц совпадает с декартовым произведением множеств точек. Обобщим операцию на m одностолбцовых таблиц:

$$\prod_{i=1\ldots m} T_i = \prod_{k=1\ldots m} \left(\prod_{j:i_j<i_k} |T_j| \cdot T_k^{\prod_{j:i_j>i_k} |T_j|} \right)$$

Пример для произведений трёх таблиц приведён в таб. **??**.

Вернёмся к направленным ограничениям. Напомню, что так называется ограничение вида $x = x(p_1, \ldots, p_n)$. В терминах таблиц эта операция соответствует добавлению столбца. Для этого нужно построить таблицу $T : I(T) = (p_1, \ldots, p_n)$. Если T содержит не все $p_{1\ldots n}$, то информации в таблице недостаточно для вычисления. Если один из столбцов таблицы, e, не является параметром функции, то можно найти подтаблицу $T' : I(T') = I(T) - \{e\}$, причём $|T'| = \frac{|T|}{|c_j|} < |T|$, что ускорит вычисления в $|c_j|$ раз. Если $p_{1\ldots n}$ — свободные переменные, таблицу можно найти как $T = \prod_{F:I(F)\subset p_{1\ldots n}} T$.

Ещё одно свойство позволяет оперировать столбцами таблицы независимо, но при этом сохранять соответствие. Будем говорить, что таблицы T_1, T_2 совместимы, если $base(T_1) = base(T_2)$, где $base(T)$ — множество свободных переменных, о которых зависят переменные из $I(T)$. Для таких таблиц действует простое правило$T_1 \times_T T_2 = T_1 \times proj(T_2, I(T_2) \setminus I(T_1))$, то есть табличное произведение эквивалентно кортежному.

Самая общая операция на таблицах — унификация. Она позволяет нам получать все комбинации значений переменных, имеющих общие зависимости. Результат унификации таблиц T_1, T_2 должен содержать все зависимости из оригинальных таблиц, значит, $base(unify(T_1, T_2)) = base(T_1) \cup base(T_2)$. Поэтому достаточно рассмотреть задачу унификации только для $S, T : base(S) \subset base(T)$. Для этого потребуется новая функция, обобщающая конкатенацию и скалярное произведение — $blockrep(b, c, d) = \sum T^c_{bi:b(i+1)}$. Легко заметить, что $blockrep(1, c, d) = c \cdot t$, $blockrep(|d|, c, d) = d^c$, $\forall b : d \div b \Rightarrow blockrep(b, 1, d) = d$. Тогда функция, совмещающая S и T:

```
Contextify(S, B): S - table, b_t - target base variable set
    b_s = base(S)
    R = S
    blockSize = 1
    for b in B
        if b not in base(S)
            R = blockrep(blockSize, |b|, R)
        blockSize = blockSize * |b|
    return R
```

Из свойств операции следует, что $\forall T : |I(t) \cap B| = 1, \forall B' \subseteq B$

$$Contextify(Contextify(T, B'), B) = Contextify(T, b)$$

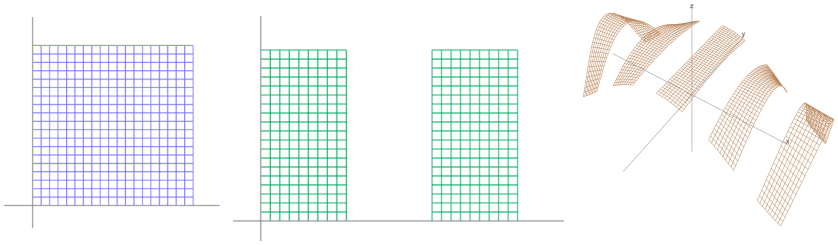
Функция действительно переводит каждый базовый столбец в целевой контекст, и, значит, работает корректно.

5.3 Топологические типы

Полученная структура графа потока имеет удачную интерпретацию в терминах прямых сумм пространств. Для каждой базовой переменной можно задать граф смежности. Это позволяет описать метрическую структуру множества значений. Например, структура дискретного множества $d \in D = \{d_1, d_n\}$ хорошо описывается пустым графом. Для конечной аппроксимации отрезка $x \in [x_0, x_1]$, $(x_0, x_1, \ldots, x_n) : x_i > x_{i-1}$ подходит граф-путь. Более сложные структуры, например, циклические или состоящие из объединения непересекающихся отрезков, тоже можно описать таким графом.

Если $x = x(p_1, \ldots, p_n), \forall p_i, p_j T(p_i) = T(p_j) = T$, причём d — непрерывная функция, то $T(d) = T$. Более практичная версия этого утверждения, позволяющая вносить исправления в граф, выглядит так:

$$E_d = \{(i, j) \in E_g | d \in \mathcal{C}(Conv(p_i, p_j))\}$$



(a) $[0,1] \times [0,1]$ (b) $[0,1] \times ([0,1] \cup [1,2])$ (c) Функция на сложной области

Рис. 5.1: Графы для различных областей

Таким образом, задача сводится к одномерному случаю. Аналогично можно определить производные структуры для случая мешей, когда смежность задана тройками вершин.

Для произведения базовых пространств логично использовать декартово произведение графов. Формально $G_1 \times G_2 = \{V_1 \times V_2, E\}, ((e_{11}, e_{12}),$ где $(e_{21}, e_{22})) \in E \Leftrightarrow (e_{11}, e_{21}) \in E_1 \vee (e_{12}, e_{22}) \in E_2$. Если использовать представление графов в GL, операцию можно описать эффективным алгоритмом:

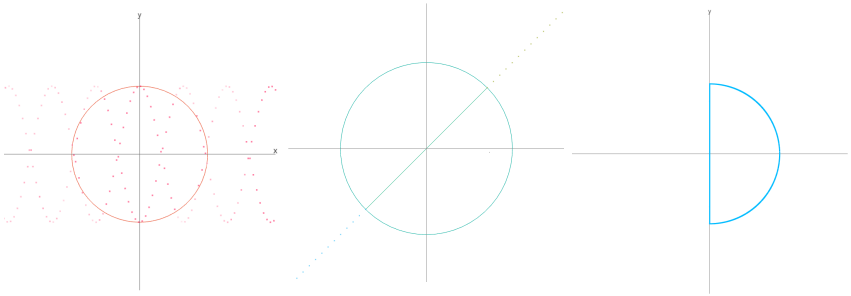
```
CartesianGraphProduct(E1, N1, Q1, E2, N2, Q2):
    E - adjacency matrix, N - node count, Q - edge count
    N = N1 * N2
    Q = N1 * Q2 + N2 * Q1
    P = []
    for i = 1..N1
        P = concat(P, E_1 + i * N_2)
    for i = 1..N2
        P = concat(P, E2 * N1 + i)
    return P
```

На рис. 5.1 приведены примеры декартовых произведений графов и области, которые они могут описывать.

Помимо возможности представлять сложные области, такие графы смежности имеют и менее очевидные, но полезные применения. Поиск всех точек, ближайших к данной, в объекте с графом смежности, происходит гораздо быстрее, чем просто во множестве точек (за $O(E)$, если наивно использовать GL-представление графа). Информацию о соседних точках можно использовать для поиска локальных экремумов (просто сравнивать значение целевой координаты с соседями), автоматического обнаружения разрывов (большой скачок значения между парой соседних точек) или для численного дифференцирования.

5.4 Решение ненаправленных ограничений

Используя материал главы, можно обрабатывать сложные системы направленных ограничений. Но не все системы можно привести к такой форме.



(a) Фильтрация (b) Продвинутая фильтрация (c) Метод Ньютона

Рис. 5.2: Методы решения ненаправленных ограничений

Далее я предполагаю, что системе доступна процедура, решающая нелинейные системы уравнений и неравенств. Это не всегда так, но на практике многомерный метод Ньютона или нелинейный вариант метода Гаусса-Зейделя дают удовлетворительные результаты.

В некоторых случаях можно обойтись без использования сложных алгоритмов. Пусть множество ограничений имеет вид $x \leq (\geq) F_{u(l)}(p_1, \ldots, p_n)$, содержит оба типа неравенств, и при этом никакое одностороннее ограничение не задаёт x. В таком случае можно легко привести систему к виду $x \in [\max\{F_l\}, \min\{F_u\}]$. Остаётся расположить x на сетке в этих пределах.

Теперь рассмотрим систему, содержащую только дискретные переменные. В такой ситуации можно считать, что данные полностью и достаточно точно представляют все возможные значения переменной. Чтобы применить такое ограничение, достаточно отфильтровать точки, не удовлетворяющие неравенству. Чтобы не удалять лишние точки, можно просто сделать их прозрачными (конечно, часть вычислений будет избыточной). Пример на рис. 5.2a.

Наконец, если среди параметров нет неопределённых значений, а все ограничения — неравенства, можно применить технику, которую я называю "псевдонепрерывной фильтрацией". Этот метод — обобщение простой фильтрации. Он состоит в том, что после фильтрации по значений программа проходит по матрице смежности объекта и ищет рёбра, пересекающие границу объекта. Для каждого такого ребра вершина, расположенная снаружи, сдвигается на границу. Для этого подойдёт любой метод поиска корней: линейная интерполяция даёт приемлемый результат, для повышения точности можно использовать метод бисекции. Для простого случая $y = x \wedge x^2 + y^2 \leq 1$ пример на рис. 5.2b.

Последний случай — вариация алгоритма "шагающие кубы". Уравнения с тремя переменными $F(x, y, z) = 0$ в общем положении задаёт поверхность. Если F гладкая, и меняет знак на поверхности, то применимы алгоритмы типа шагающих кубов. Для двумерного случая существует аналогичный метод шагающих квадратов. Идея метода очень проста: область разбивает-

ся на квадраты и в каждой вершине вычисляется значение функции. Если в вершинах квадрата функция принимает значения противоположных знаков, то линия уровня пересекает его рёбра.

Метод шагающих квадратов также не всегда применим. Если область ограничена несколькими неравенствами, то граница, скорее всего, не будет гладкой. В таком случае можно свернуть неравенства в одно: $F(x, y) \leq 0 \wedge G(x, y) \leq 0$ переходит в $\max(F(x, y), G(x, y)) \leq 0$. После этого граница эффективно ищется многомерным методом Ньютона. Пример для системы $x \geq 0 \wedge x^2 + y^2 \leq 1$ — рис. 5.2c. Обратите внимание, что острые точки, в которых пересекаются ограничения, сохранились.

Остаётся определиться с тем, какой вид должны принимать ограничения в структуре графа потока данных. Как я уже говорил, двустороннее ограничение имеет обратную силу, то есть влияет на переменные, находящиеся ниже него в графе потока данных. Это противоречит идее графа потока и может привести к неправильному результату вычислений. Чтобы избежать таких неприятностей, нужно замкнуть каждое неявное ограничение относительно предшествования в графе потока. Разумеется, это нужно сделать перед разложением в прямое произведение.

Циклические зависимости можно привести к такому же виду. Пусть $C \subseteq V$ — компонента сильной связности графа $G = \langle V, E \rangle$. Иначе говоря, между любой парой вершин в C существует путь. Любая нетривиальная (содержащая более одной вершины) компонента связности содержит хотя бы один цикл, и такой граф потока не является корректным. Компоненту связности можно описать эквивалентной системой уравнений, то есть ненаправленным ограничением. Так, система $y = y(x) \wedge x = x(y)$ не является допустимым выражением в направленных ограничениях, но разрешима как система уравнений.

5.5 Выводы

Для решения систем ограничений потребовалось обобщить механизм реактивного программирования. Из структуры зависимостей между переменными можно извлечь информацию для оптимизации работы системы. Основные операции при решении систем ограничений — вычисление прямого произведения и отображение. В первой части главы я строю формальную модель операций на таблицах данных. Самая общая операция, унификация, позволяет объединить две таблицы так, чтобы получить все возможные комбинации их элементов. Также я предлагаю механизм хранения структурной информации в виде графа и определяю действие табличных операторов на таком представлении. Наконец, я описываю несколько способов решения ненаправленных ограничений и условия их применимости.

6 Заключение

В данной работе представлен набор методов, применимых для создания системы математической визуализации. Такая система полезна как для практикующих математиков, так и для студентов. Важные требования к системе — возможность создания трёхмерных изображений, кроссплатформенность и интерактивность. Ни одно из существующих решений не удовлетворяет таким требованиям.

Я рассматриваю известные формальные модели визуализации и строю четырёхступенчатую декомпозицию процесса компьютерной визуализации математических объектов. Процесс покрывает полный цикл, начиная от символьной формы объекта и заканчивая взаимодействием пользователя с изображением.

Системы ограничений — мощное и понятное средство описания математических объектов. Пользователю системы удобно задавать объекты в такой форме, и формулы можно автоматически транслировать в программы. Модель используется как для непосредственного взаимодействия с пользователем, так и для создания программной архитектуры системы. Этот механизм низкого (с точки зрения математики) уровня гибок и прост в реализации. Я выделяю три оператора, необходимых для программной реализации такой модели: разложение в прямое произведение, выделение направленных ограничений и численное решение решение систем зависимостей.

Главное нововведение работы — приложение парадигмы реактивного программирования к визуализации. Реактивное программирование — декларативный метод, подходящий для создания систем реального времени и концептуально близкий к предметной области. Я показываю, что для типичного паттерна взаимодействия с системой визуализации в худшем случае оптимальна pull-реактивная реализация с кешированием результатов вычислений.

В главе 5 подробнее анализируется связь реактивных моделей с системами ограничений. Обычно реактивные системы анализируют в моделях цифровой обработки сигналов: переменные — числа, зависящие от времени. Для многомерной визуализации полезнее анализировать не зависимость переменных от времени, а взаимосвязи между переменными. Я предлагаю модель данных, которая хорошо сочетается с основными операторами на системах ограничений и устраняет избыточные вычисления. В математической визуализации часто доступны данные о метрической структуре пространств, в которых лежат переменные. Такие данные можно эффективно представить графом смежности. Действие декартова произведение таблиц и ненаправленных ограничений эффективно представляются соответствующими операциями на графах смежности. Наконец, я описываю набор численных методов, эффективно разрешающих ненаправленные и циклические зависимости в графах потока данных.

6.1 Направления дальнейших исследований

Кратко приведу список вопросов, достойных дальнейшего изучения.

- Системы ограничений в действительных числах — мощный инструмент, но в современной математике приняты и более общие способы задания объектов. Дальнейшие исследования в этой области приведут к стандартизированной математической нотации и разработке универсальных математических яыков программирования. Более близкие цели для библиотеки — поддержка дифференциальных и теоретико-множественных операторов и на уровне программного интерфейса.

- Реактивное программирование допускает различные реализации. Сейчас библиотека всегда использует pull-реактивную модель с кешированием результатов вычислений. Такая модель оптимальна для худшего случая. Более подробный статистический анализ работы системы позволил бы переходить на более эффективные модели, когда это необходимо. Информацию о структуре графа зависимостей можно использовать для автоматической параллелизации программ или делегации вычислений графическому процессору.

- Когда математик строит график вручную, он редко следует нашей модели процесса. Обычно человек выделяет несколько существенных особенностей объекта и изображает их, уделяя мало внимания остальным свойствам. Особенностями могут быть связность области, монотонность или периодичность функции, существование локального экстремума. Если система сможет выделять такие существенные факты об объекте, результат визуализации будет достижим существенно меньшим числом точек.

Список литературы

[1] Jacques Bertin. Sémiologie graphique: Les diagrammes-les réseaux-les cartes. 1973.

[2] David M Butler and MH Pendley. A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, 3(5):45–51, 1989.

[3] William S Cleveland et al. *The elements of graphing data*. Wadsworth Advanced Books and Software Monterey, CA, 1985.

[4] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. *Combinatory logic*, volume 2. North-Holland Amsterdam, 1972.

[5] Stephen G Eick and Graham J Wills. High interaction graphics. *European Journal of Operational Research*, 81(3):445–459, 1995.

[6] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

[7] Conal M Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM, 2009.

[8] Miguel de Guzman. The role of visualization in the teaching and learning of mathematical analysis. 2002.

[9] Robert B Haber and David A McNabb. Visualization idioms: A conceptual model for scientific visualization systems. *Visualization in scientific computing*, 74:93, 1990.

[10] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[11] William L Hibbard, Charles R Dyer, and Brian E Paul. A lattice model for data display. In *Proceedings of the conference on Visualization'94*, pages 310–317. IEEE Computer Society Press, 1994.

[12] David Hilbert and Stephan Cohn-Vossen. *Geometry and the Imagination*, volume 87. American Mathematical Soc., 1999.

[13] Zekeriya Karadag. Improving online mathematical thinking. In *11th International Congress on Mathematical Thinking in Elementary and advanced Mathematics, Educational Studies in Mathematics*, volume 38, pages 111–113, 2011.

[14] Jock Mackinlay. Automating the design of graphical presentations of relational information. *Acm Transactions On Graphics (Tog)*, 5(2):110–141, 1986.

[15] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64. ACM, 2002.

[16] Kay O'Halloran. *Mathematical discourse: Language, symbolism and visual images*. A&C Black, 2008.

[17] George Polya. *How to solve it: A new aspect of mathematical method*. Princeton university press, 2014.

[18] Hikmet Senay and Eve Ignatius. *Rules and principles of scientific data visualization*. Institute for Information Science and Technology, Department of Electrical Engineering and Computer Science, School of Engineering and Applied Science, George Washington University, 1990.

[19] Leon Sterling, Alan Bundy, Lawrence Byrd, Richard O'Keefe, and Bernard Silver. *Solving symbolic equations with PRESS*. Springer, 1982.

[20] Edward R Tufte and PR Graves-Morris. *The visual display of quantitative information*, volume 2. Graphics press Cheshire, CT, 1983.

[21] Paul Vickers, Joe Faith, and Nick Rossiter. Understanding visualization: A formal approach using category theory and semiotics. *Visualization and Computer Graphics, IEEE Transactions on*, 19(6):1048–1061, 2013.

[22] Caroline Ziemkiewicz and Robert Kosara. Understanding information visualization in the context of visual communication. Technical report, Technical Report CVCUNCC-07-08, 2007.

[23] Walter Zimmermann and Steve Cunningham. Editors' introduction: What is mathematical visualization. *Visualization in teaching and learning mathematics*, pages 1–7, 1991.

A Примеры использования библиотеки

Эта глава содержит 11 визуализаций, созданных в библиотеке grafar. Примеры разделены на три группы. Слева в примерах указана система ограничений, справа находится изображение. Снизу приведён код, создающий визуализацию. Код написан для стабильной версии библиотеки, grafar 2.7. Статические иллюстрации не могут предать интерактивность визуализаций, но это лучшее, что позволяет формат.

A.1 Базовое использование

$$\begin{cases} x \in [-2, 2] \\ y = \sin(x) \end{cases}$$

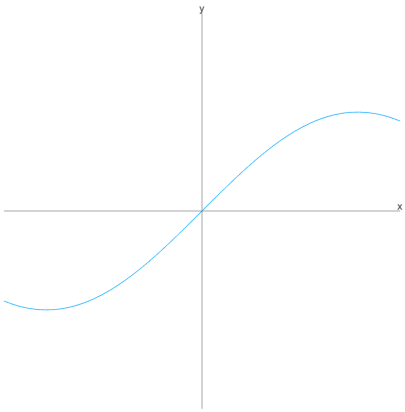


Рис. A.1: Явная функция одной переменной

```
graph.constrain({what: 'x', maxlen: 50, as: grafar.seq(-2, 2)})
    .constrain({what: 'y', using: 'x', as: function(x, y, ans) {
        ans[0] = Math.sin(x);
    }});
```

$$\begin{cases} t \in [-5, 2] \\ x = \sin(7t)e^t \\ y = \cos(7t)e^t \end{cases}$$

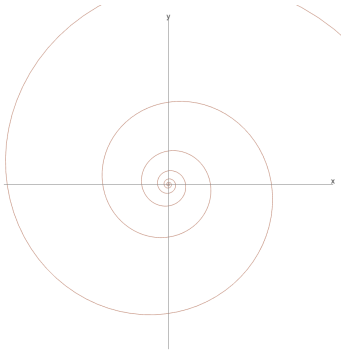


Рис. А.2: Параметрическая кривая на плоскости

```
graph.constrain({what: 't', maxlen: 500, as: grafar.seq(-5, 1.1)})
    .constrain({what: 'x', using: 't', as: function(t, ans) {
        ans[0] = Math.sin(t * 7) * Math.exp(t);
    }})
    .constrain({what: 'y', using: 't', as: function(t, ans) {
        ans[0] = Math.cos(t * 7) * Math.exp(t);
    }});
```

$$x^3 + y^2 = 2$$

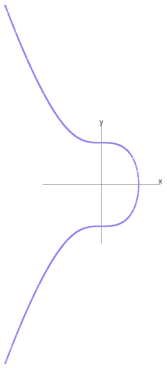


Рис. А.3: Неявно заданная кривая на плоскости

```
graph.constrain({what: 'x,y', maxlen: 5000, as: grafar.traceZeroSet
    (function(v) {
        return v[0] * v[0] * v[0] + v[1] * v[1] - 2;
    }, ['x', 'y'])
});
```

$$x^2 \sin(x) + y^2 = 2$$

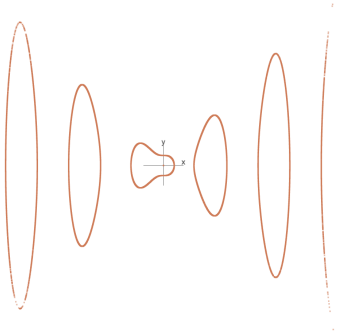


Рис. А.4: Семейство кривых на плоскости

```
graph.constrain({what: 'x,y', maxlen: 5000, as:
    grafar.traceZeroSet(function(v) {
        return v[0] * v[0] * Math.sin(v[0]) + v[1] * v[1] - 2;
    }, ['x', 'y'])
});
```

$$\begin{cases} x \in [-10, 10] \\ y \in [-10, 10] \\ z = \frac{3\sin(x)\cos(y)}{x} \end{cases}$$

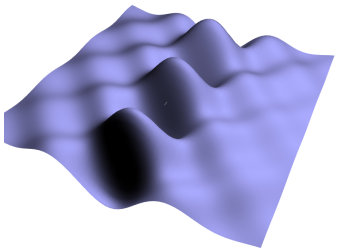


Рис. А.5: Явная поверхность

```
graph.constrain({what: 'x', maxlen: 100, as: grafar.seq(-10, 10)})
    .constrain({what: 'y', maxlen: 100, as: grafar.seq(-10, 10)})
    .constrain({what: 'z', using: 'x,y', as: function(x, y, ans) {
        ans[0] = 3 * Math.sin(x) * Math.cos(y) / x;
    }});
```

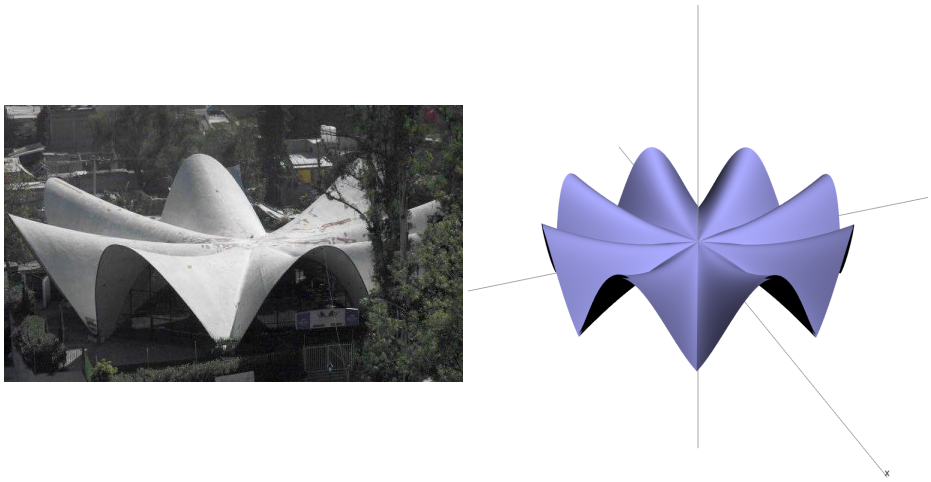


Рис. А.6: Трёхмерная модель со многими параметрами

```
var segAng = 3 * Pi / 2,
    seg = 8,
    angRescale = 2 * Pi / segAng / seg,
    pow = 2,
    h = 1 / 3;

bld1.constrain({what: 'r', maxlen: 30, as: grafar.seq(0, 1)})
    .constrain({what: 'phi', maxlen: 50, as:
        grafar.seq(-segAng / 2, segAng / 2)})
    .constrain({what: 'rep', maxlen: seg, as: grafar.dseq(0,2*Pi)})
    .constrain({what: 'layer', maxlen: 2, as: grafar.dseq(0, .02)})
    .constrain({what: 'x,y,z', using: 'r,phi,rep,layer', as:
        function(r, phi, rep, layer, ans) {
            ans[0] = Math.cos(rep + angRescale * phi) * r;
            ans[1] = Math.sin(rep + angRescale * phi) * r;
            ans[2] = h * Math.pow(r, pow) * Math.cos(phi) + layer;
    }});
```

A.2 Продвинутые примеры

Визуализации, приведённые в этой части, показывают неочевидные использования библиотеки. Хорошо, что библиотека позволяет делать больше, чем предполагалось при её создании. Но в будущем функциональность стоит наращивать, отталкиваясь именно от таких примеров.

$$\begin{cases} s \in \{-1, 1\} \\ p \in [-2, 2] \\ q \in [-2, 2] \\ n = \sqrt{1 + q^2} \\ x = q + \frac{s}{n} \\ y = p + \frac{sq}{n} \end{cases}$$

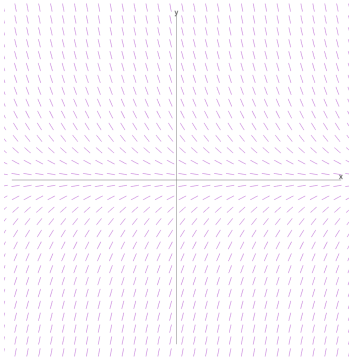


Рис. A.7: Поле направлений

$$\begin{cases} \epsilon = 10^{-5} \\ i \in \{-5, -4, \ldots, 5\} \\ t \in \left[-\frac{\pi}{2} + \epsilon, \frac{\pi}{2} - \epsilon\right] \\ x = i\pi + t \\ y = \tan x \end{cases}$$

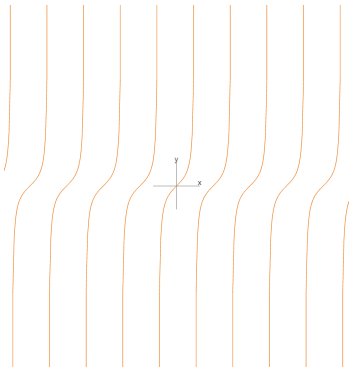


Рис. A.8: Разрывная функция через индексированный домен

A.3 Визуализации проекта VisualMath

Проект VisualMath использует grafar, реализацию системы. Идея проекта — создание коллекции динамических визуальных модулей, которые можно использовать во время лекций и самостоятельно. Сейчас проект содержит два набора визуализаций на grafar: тройные интегралы и метод множителей Лагранжа. Сами программы слишком велики, чтобы приводить их здесь, поэтому я ограничусь снимками экрана и краткими описаниями.

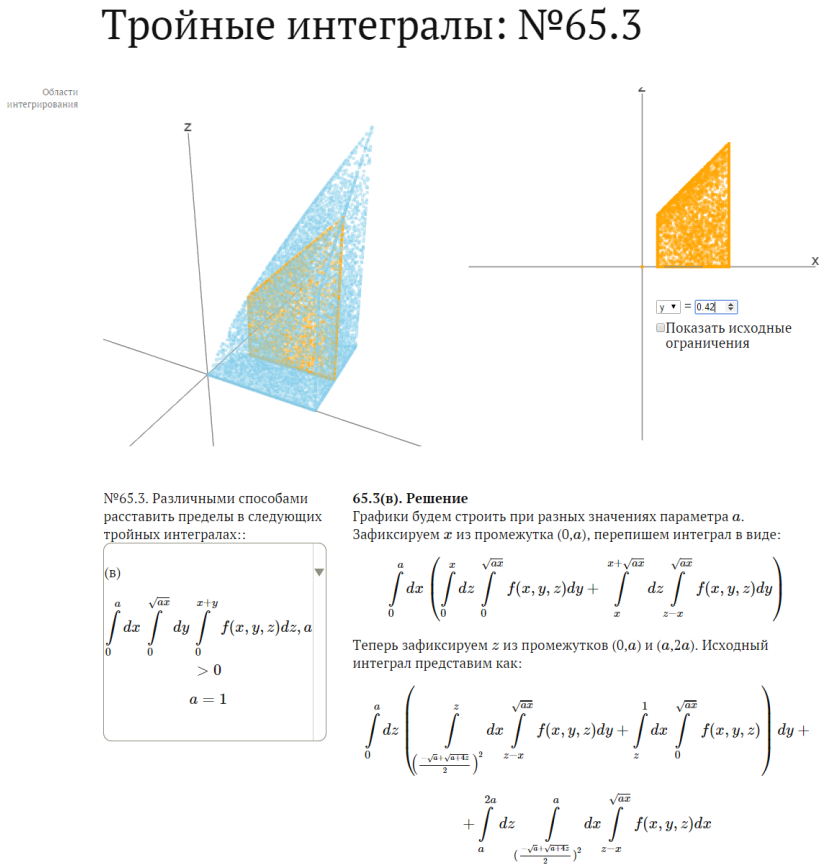


Рис. А.9: Тройные интегралы

Пользователь видит трёхмерную область, по которой вычисляется интеграл. На втором графике изображено сечение области плоскостью. Выбирая разные оси и меняя систему координат можно найти более простое представление объекта. Для каждой задачи приведено аналитическое решение. Также программа использует Eulerface, набор шаблонов и элементов интерфейса для онлайн-системы математического образования.

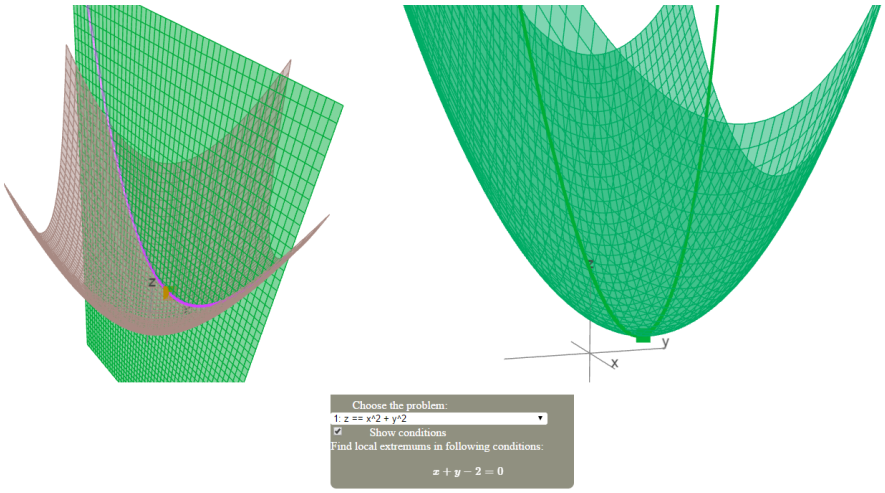


Рис. А.10: Метод множителей Лагранжа

Метод множителей Лагранжа сложен для понимания. Эта программа пытается сделать тему понятнее. На левом графике студент видит целевую функцию и условие, при котором она оптимизируется. Оптимальные точки при ограничении — экстремумы кривой, находящейся на пересечении поверхностей. На правом графике изображены функции Лагранжа — поверхности, экстремумы которых находятся в целевых точках.