Міністерство освіти і науки України

Харківський національний університет імені В.Н. Каразіна

Кафедра комп'ютерної фізики

"**ЗАТВЕРДЖУЮ**"

Завідувач кафедри

_____ проф. Костянтин НЄМЧЕНКО

"_____"__ _____2020р.

Навчальний контент навчальної дисципліни

МОДЕЛЮВАННЯ ЕНЕРГЕТИЧНИХ ПРОЦЕСІВ

рівень вищої освіти	другий (магістерський)
галузь знань	10 Природничі науки
спеціальність	105 Прикладна фізика та наноматеріали
освітня програма	«Прикладна фізика енергетичних систем»
вид дисципліни	вибіркова
навчально – науковий інститут	комп'ютерної фізики та енергетики

2020 / 2021 навчальний рік

1. Опис навчальної дисципліни

Предметом вивчення навчальної дисципліни є сучасні методи комп'ютерної симуляції ї для задач фізики та енергетики, розробка та використання існуючих бібліотек математичних методів обробки даних.

1.1. Мета викладання навчальної дисципліни

Метою курсу «Моделювання енергетичних процесів» є вивчення та самостійне користування алгоритмами обробки даних з використанням самостійно розроблених алгоритмів та бібліотек для задач фізики та математики.

1.2. Основні завдання вивчення дисципліни

Основним завданням курсу «Моделювання енергетичних процесів» є застосування математичних алгоритмів для обробки багатовимірних масивів даних, зокрема, зображень, для низки задач, що є типовими для фізики та енергетики.

У результаті вивчення даного курсу студент повинен

знати: методи комп'ютерної симуляція в фізиці за допомогою алгоритмів, які реалізовані на сучасних мовах програмування та самостійне створення програмних продуктів , що розв'язують певні задачі в галузі фізики та енергетики.

вміти: застосовувати отримані знання на практиці при обробці багатовимірних масивів даних, узагальнювати вивчені алгоритмі на складні системи.

Для вивчення курсу необхідні знання з програмування, теорії алгоритмів, математичного аналізу, диференціальних рівнянь, математичної фізики, та курсів лінійної алгебри та аналітичної геометрії. Основними формами викладання навчального матеріалу з дисципліни «Моделювання енергетичних процесів» є лекції, практичні заняття та самостійна робота студентів.

1.3. Кількість кредитів 5

1.4. Загальна кількість годин 150

1.5. Характеристика навчальної дисципліни	
За вибором	
Денна форма навчання	Заочна (дистанційна) форма навчання
Рік підготовки	
1-й	-й
Семестр	
1-й	-й
Лекції	
год.	год.
Лабораторні заняття	
48 год.	год.
Самостійна робота	
102 год.	год.

1.6. Заплановані результати навчання

В результаті вивчення навчальної дисципліни студенти оволодіють сучасними комп'ютерної симуляції в фізиці енергетики, а також засобами розробки та використання існуючих бібліотек математичних методів обробки даних.

2. Тематичний план навчальної дисципліни

Розділ 1. Стандартні бібліотеки шаблонів

Тема 1. Вступ до STL

Ідеологія бібліотеки стандартних шаблонів

Тема 2. Використання STL в обробці даних

Типи даних в STL. Доступ до даних.

Тема 3. Використання STL для розробки алгоритмів

Стандартні абстрактні алгоритми.

Розділ 2. Паралельні обчислення

Тема 1. Вступ до методів паралельних обчислень

Ідеологія паралелізації процесів обчислення.

Тема 2. Реалізація в певних мовах програмування

Методи паралелізації для конкретних програм.

Тема 3. Паралелізація алгоритмів в фізиці

Використання паралельних обчислень у моделювання фізичних процесів.

Метод Монте-Карло. Керування експериментом.

Розділ 3. Стандартні бібліотеки алгоритмів

Тема 1. Бібліотеки математичних примітивів

Поняття математичних примітивів. Використання примітивів в моделюванні.

Тема 2. Бібліотеки обробки сигналів

Використання бібліотек обробки сигналів для обробки даних в фізиці.

Тема 3. Бібліотеки обробки зображень

Використання стандартних бібліотек для обробки даних візуалізації в енергетиці.

3. Теми лабораторних занять

№ з/п	Назва теми	Кількість годин
1	Типи контейнерів	6
2	Доступ до даних	6
3	Стандартні абстрактні алгоритми	4
4	Паралелізації процесів обчислення.	6
5	Керування експериментом.	6
6	Метод Монте-Карло	4
7	Використання примітивів в моделюванні.	4
8	Використання бібліотек обробки сигналів в фізиці.	6
9	Використання стандартних бібліотек для обробки даних	6
	Усього	48

4. Завдання для самостійної робота

№ з/п	Назва теми	Години
1	Вивчити типи контейнерів	11
2	Дослідити доступ до даних	11
3	Ознайомитись зі стандартними абстрактними алгоритмами	12
4	Дослідити паралелізацію процесів обчислення.	12
5	Навчитися використовувати керування експериментом.	12
6	Ознайомитись з методом Монте-Карло	10
7	Ознайомитись з використанням примітивів в моделюванні.	11
8	Ознайомитись з використанням бібліотек обробки сигналів в фізиці.	12
9	Ознайомитись з використанням стандартних бібліотек для обробки даних	11
	Усього	102

5. Навчальний контент частина 1

1. Комп'ютерна симуляція (комп’ютерне моделювання) – використання комп'ютера для представлення динамічних відповідей однієї системи поведінкою іншої системи, змодельованої за нею. Моделювання використовує математичний опис або модель реальної системи у вигляді комп'ютерної програми. Ця модель складається з рівнянь, які дублюють функціональні зв'язки всередині реальної системи. Після запуску програми отримана математична динаміка формує аналог поведінки реальної системи, результати якої представлені у вигляді даних. Моделювання також може мати форму комп'ютерно-графічного зображення, яке представляє динамічні процеси в анімованій послідовності.

2. Комп'ютерне моделювання використовується для вивчення динамічної поведінки об'єктів або систем у відповідь на умови, які неможливо легко або безпечно застосувати в реальному житті. Наприклад, ядерний вибух може бути описаний математичною моделлю, яка включає такі змінні, як тепло, швидкість і радіоактивні викиди. Потім можуть бути використані додаткові математичні рівняння, щоб пристосувати модель до змін певних змінних, таких як кількість розщеплюваного матеріалу, який виробляв вибух. Симуляції особливо корисні для того, щоб дозволити спостерігачам виміряти і передбачити, як на роботу цілої системи може вплинути зміна окремих компонентів у цій системі.

3. Простіше моделювання, що проводиться персональними комп'ютерами, складається в основному з бізнес-моделей і геометричних моделей. Перша включає в себе електронні таблиці, фінансові та статистичні програми, які використовуються у бізнес-аналізі та плануванні. Геометричні моделі використовуються для численних застосувань, які вимагають простого математичного моделювання об'єктів, таких як будівлі, промислові частини, і молекулярні структури хімічних речовин. Більш просунуті симуляції, такі як ті, що імітують погодні умови або поведінку макроекономічних систем, зазвичай виконуються на потужних робочих станціях або на ЕОМ. У машинобудуванні комп'ютерні моделі новостворених конструкцій піддаються моделюванню тестів для визначення їхніх реакцій на напругу та інші фізичні змінні. Моделювання річкових систем можна маніпулювати, щоб визначити потенційний вплив гребель та зрошувальних мереж перед будь-яким фактичним будівництвом. Інші приклади комп'ютерного моделювання включають оцінку конкурентних реакцій компаній на певному ринку та відтворення руху та польоту космічних апаратів.

КС в порівнянні з моделлю

Комп'ютерною моделлю є алгоритми і рівняння, що використовуються для відображення поведінки системи, яка моделюється. На відміну від цього, КС є фактичним запуском програми, яка містить ці рівняння або алгоритми. КС, отже, є процесом запуску моделі. Таким чином, не можна було б «побудувати симуляцію»; замість цього можна було б "побудувати модель", а потім або "запустити модель" або еквівалентно "запустити КС".

Підготовка даних

Зовнішні вимоги до даних для КС та моделей широко варіюються. Для деяких вхідних даних може бути лише кілька цифр (наприклад, моделювання сигналу електричної енергії змінного струму на дроті), тоді як інші можуть вимагати терабайт інформації (наприклад, моделі погоди та клімату).

Джерела вхідних даних також дуже відрізняються:

1. Датчики та інші фізичні пристрої, підключені до моделі;
2. Поверхні, що використовуються для направлення ходу моделювання певним чином;
3. Поточні або історичні дані, введені вручну;

4. Значення, отримані як побічний продукт від інших процесів;
5. Значення виводяться для цілей іншими КС, моделями або процесами.

Нарешті, час, коли дані доступні, змінюється:

– "інваріантні" дані часто вбудовуються в код моделі, або тому, що значення є дійсно інваріантним (наприклад, значенням π), або тому, що розробники вважають значення інваріантним для всіх випадків, що представляють інтерес;

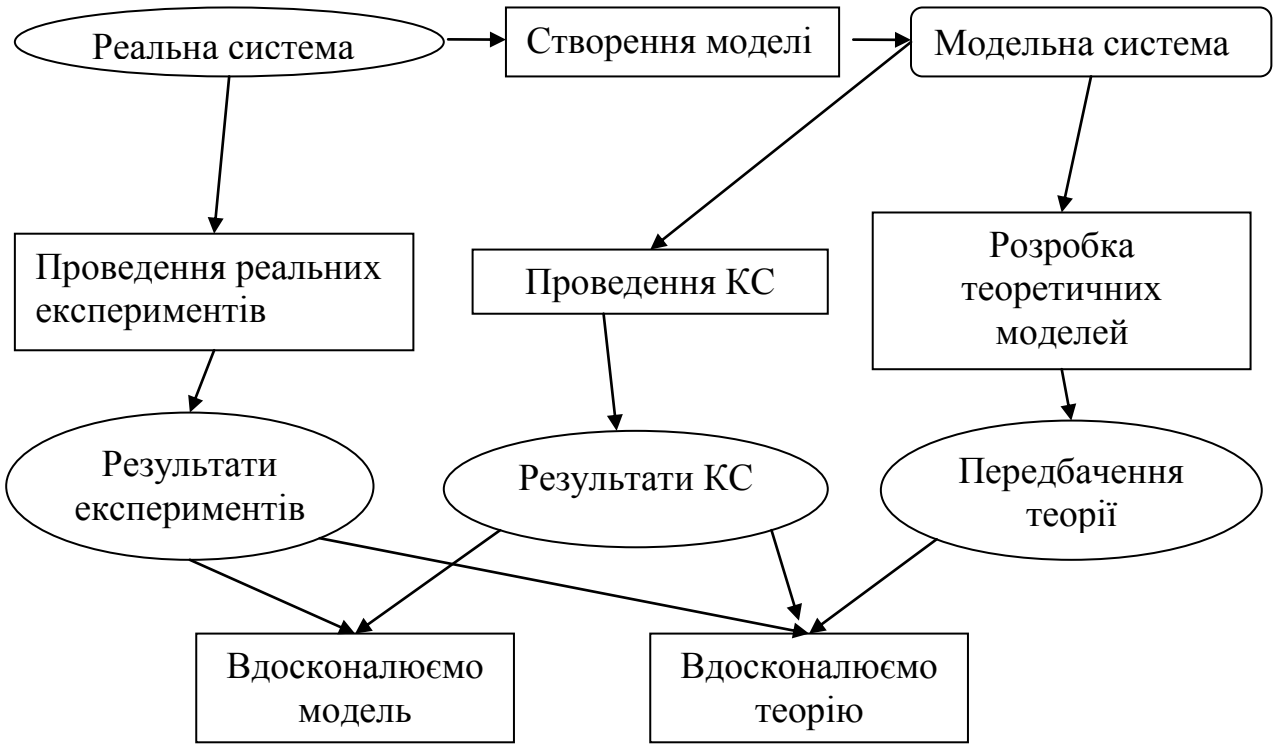
– дані можна вводити в симуляцію, коли він запускається, наприклад, зчитуючи один або більше файлів, або зчитуючи дані з препроцесора;

– дані можуть бути надані під час моделювання, наприклад, мережею датчиків.

Системи, які приймають дані з зовнішніх джерел, повинні бути дуже «обережними» у знанні того, що вони отримують. Хоча комп'ютери легко читають у значеннях з текстових або бінарних файлів, що набагато важче знати, яка точність (порівняно з роздільною здатністю і точністю вимірювання) є значеннями. Часто вони виражаються як "смуги помилок", мінімальне і максимальне відхилення від діапазону значень, в межах яких лежить справжнє значення (очікується). Оскільки математична обчислювальна техніка не є досконалою, помилки округлення і обрізання помножують цю помилку, тому корисно виконати "аналіз помилок", щоб підтвердити, що значення виходу моделювання все ще будуть корисними.

Навіть невеликі помилки у вихідних даних можуть накопичуватися в істотну помилку пізніше в симуляції. Хоча весь комп'ютерний аналіз підлягає обмеженню "GIGO" (сміття зайшло – сміття вийшло), це особливо стосується цифрового моделювання. Дійсно, спостереження цієї невід'ємної кумулятивної помилки в цифрових системах було головним каталізатором розвитку теорії хаосу.

Процес побудови комп'ютерної моделі, а також взаємодія експерименту, моделювання та теорії.



Типи КС

Комп'ютерні моделі можна класифікувати за кількома незалежними **парами** атрибутів, у тому числі:

Стохастичний або детермінований (і як окремий випадок детермінованого - хаотичного)

Стаціонарний або динамічний

Безперервний або дискретний

Симуляція динамічної системи, напр. електричні системи, гідравлічні системи або багатокомпонентні механічні системи (описані в першу чергу за допомогою DAE: s) або динамічне моделювання польових задач, напр. CFD моделювання FEM (описано PDE: s).

Місцеві або розподілені.

Іншим способом класифікації моделей є перегляд основних структур даних. Для моделювання з кроком у часі існує два основних класи:

Симуляції, які зберігають свої дані в регулярних сітках і вимагають лише доступу до сусідніх сусідів, називаються кодами трафарету. Багато додатків CFD належать до цієї категорії.

Якщо основний графік не є регулярною сіткою, модель може належати до методу безсистемного методу.

Рівняння визначають взаємозв'язки між елементами модельованої системи і намагаються знайти стан, в якому система знаходиться в рівновазі. Такі моделі часто використовуються у моделюванні фізичних систем, як простіший випадок моделювання до спроби динамічного моделювання.

Динамічні моделі моделювання змінюються в системі у відповідь на (зазвичай змінюються) вхідні сигнали.

Стохастичні моделі використовують генератори випадкових чисел для моделювання випадкових або випадкових подій;

Моделювання дискретних подій (DES) управляє подіями в часі. Більшість комп'ютерних, логіко-тестових і дерев моделювання відмов такого типу. У цьому типі моделювання симулятор зберігає чергу подій, відсортованих за імітованим часом, вони повинні відбуватися. Симулятор читає чергу і запускає нові події, коли обробляється кожна подія. Не важливо виконувати моделювання в режимі реального часу. Часто важливіше мати доступ до даних, отриманих в результаті моделювання, і виявляти логічні дефекти в дизайні або послідовності подій.

Безперервне динамічне моделювання виконує чисельне рішення диференціально-алгебраїчних рівнянь або диференціальних рівнянь (часткових або звичайних). Періодично програма моделювання вирішує всі рівняння і використовує числа для зміни стану і виходу моделювання. Додатки включають імітатори польоту, ігри для моделювання конструкцій та управління, моделювання хімічних процесів і моделювання електричних ланцюгів. Спочатку ці типи моделювання були фактично реалізовані на аналогових комп'ютерах, де диференціальні рівняння могли бути представлені безпосередньо різними електричними компонентами, такими як оп-ампер. До кінця 1980-х років, проте, більшість "аналогових" симуляцій виконувалося на звичайних цифрових комп'ютерах, які імітують поведінку аналогового комп'ютера.

Спеціальний тип дискретного моделювання, який не спирається на модель з базовим рівнянням, але, тим не менш, формально може бути представлений, є імітацією на основі агентів. У моделюванні на основі агентів окремі сутності (такі як молекули, клітини, дерева або споживачі) в моделі представлені безпосередньо (а не їх щільністю або концентрацією) і

мають внутрішній стан і набір поведінок або правил, які визначають, як Стан агента оновлюється з одного кроку в інший.

Розподілені моделі працюють на мережі взаємопов'язаних комп'ютерів, можливо, через Інтернет. Симуляції, рознесені на декількох хост-комп'ютерах, як це, часто називають "розподіленими симуляціями". Існує декілька стандартів для розподіленого моделювання, включаючи протокол моделювання сукупного рівня (ALSP), розподілене інтерактивне моделювання (DIS), архітектуру високого рівня (моделювання) (HLA) і архітектуру включення тестування і навчання (TENA).

Візуалізація

Раніше вихідні дані комп'ютерного моделювання іноді були представлені в таблиці або в матриці, що показує, як на дані впливали численні зміни параметрів моделювання. Використання матричного формату було пов'язане з традиційним використанням матричної концепції в математичних моделях. Тим не менш, психологи та інші відзначили, що люди можуть швидко сприймати тенденції, розглядаючи графіки або навіть рухомі зображення або картинки, що генеруються з даних, які відображаються анімацією зображень генерованих комп'ютером (CGI). Незважаючи на те, що спостерігачі не завжди зможуть читати цифри або формулювати математичні формули, спостерігаючи за графіком погоди, вони могли б передбачити події (і "побачити, що дощ вказував їм шлях") набагато швидше, ніж сканування таблиць координат дощових хмар. Такі інтенсивні графічні дисплеї, які виходять за межі світу чисел і формул, іноді також призводять до виходу, на якому відсутня координатна сітка або пропущені часові мітки, як якщо б занадто далеко від дисплеїв числових даних. Сьогодні, моделі прогнозування погоди, як правило, врівноважують погляди на переміщення дощових / снігових хмар на карту, яка використовує числові координати і числові часові мітки подій.

Аналогічно, комп'ютерне моделювання CGI-сканувань CAT може імітувати, як пухлина може зменшуватися або змінюватися протягом тривалого періоду медичного лікування, представляючи проходження часу як вид обертання видимої людської голови, коли змінюється пухлина.

Інші програми CGI комп'ютерного моделювання розробляються для графічного відображення великих обсягів даних в русі, оскільки зміни відбуваються під час моделювання.

Комп'ютерне моделювання в науці

Загальні приклади типів комп'ютерного моделювання в науці, які виводяться з основного математичного опису:

чисельна КС диференціальних рівнянь, які не можуть бути вирішені аналітично, теорії, що включають такі безперервні системи, як явища у фізичній космології, динаміка рідини (наприклад, кліматичні моделі, моделі шуму на дорожньому шляху, моделі дисперсії повітряних доріг), механіка континууму та хімічна кінетика.

стохастична КС, як правило, використовується для дискретних систем, де події відбуваються ймовірно і які не можуть бути описані безпосередньо з диференціальними рівняннями (це дискретне моделювання у вищевказаному сенсі). Явища цієї категорії включають генетичні дрейфи, біохімічні або генні регуляторні мережі з невеликою кількістю молекул. (див. також: метод Монте-Карло).

багаточастинна КС реакції наноматеріалів на множинні масштаби до прикладеної сили з метою моделювання їх термопружних і термодинамічних властивостей. Методики, що

використовуються для таких моделювань, - молекулярна динаміка, молекулярна механіка, метод Монте-Карло, багато масштабна функція Гріна.

Нижче наведено конкретні приклади комп'ютерних моделювань:

Статистична КС, засноване на агломерації великої кількості вхідних профілів, таких як прогнозування рівноважної температури приймаючих вод, що дозволяє вводити гаму метеорологічних даних для конкретної мови. Ця методика розроблена для прогнозування теплового забруднення.

Моделювання на основі агентів ефективно використовувалося в екології, де його часто називають "індивідуальним моделюванням" і використовують у ситуаціях, коли індивідуальну мінливість агентів не можна нехтувати, наприклад, динаміку популяції лосося і форелі (більшість суто математичних моделей припускають: вся форель поводиться однаково).

Часова динамічна модель. У гідрології існує кілька таких транспортних моделей гідрології, як моделі SWMM і DSSAM, розроблені агентством охорони навколишнього середовища США для прогнозування якості річкової води.

КС також використовувалося для формального моделювання теорій пізнання і продуктивності людини, наприклад, ACT-R.

КС з використанням молекулярного моделювання для виявлення лікарських засобів. [10]

КС для моделювання вірусної інфекції в клітинах ссавців. [9]

Для КС поведінки потоку повітря, води та інших рідин використовуються обчислювальні імітаційні методики рідинної динаміки. Використовуються одно-, дво- і тривимірні моделі. Одновимірна модель могла б імітувати вплив гідравлічного удару в трубі. Двовимірна модель може бути використана для імітації сил опору на поперечному перерізі крила літака. Тривимірне моделювання може оцінити потреби опалення та охолодження великої будівлі.

Розуміння статистичної термодинамічної молекулярної теорії є фундаментальним для оцінки молекулярних рішень. Розробка теореми розподілу потенціалу (PDT) дозволяє спростити цю складну тему до презентацій молекулярної теорії.

Прикладні пакети для КС для фізики та техніки

Розроблено графічні середовища для моделювання. Особливу увагу було приділено обробці подій (ситуації, в яких імітаційні рівняння не є дійсними і повинні бути змінені). Відкритий проект «Open Source Physics» розпочав розробку багаторазових бібліотек для моделювання в Java, а також Easy Java Simulations - повне графічне середовище, що генерує код на основі цих бібліотек.

6. Навчальний контент частина 2

Основні джерела

1. The Ultimate Guide to JavaScript Physics

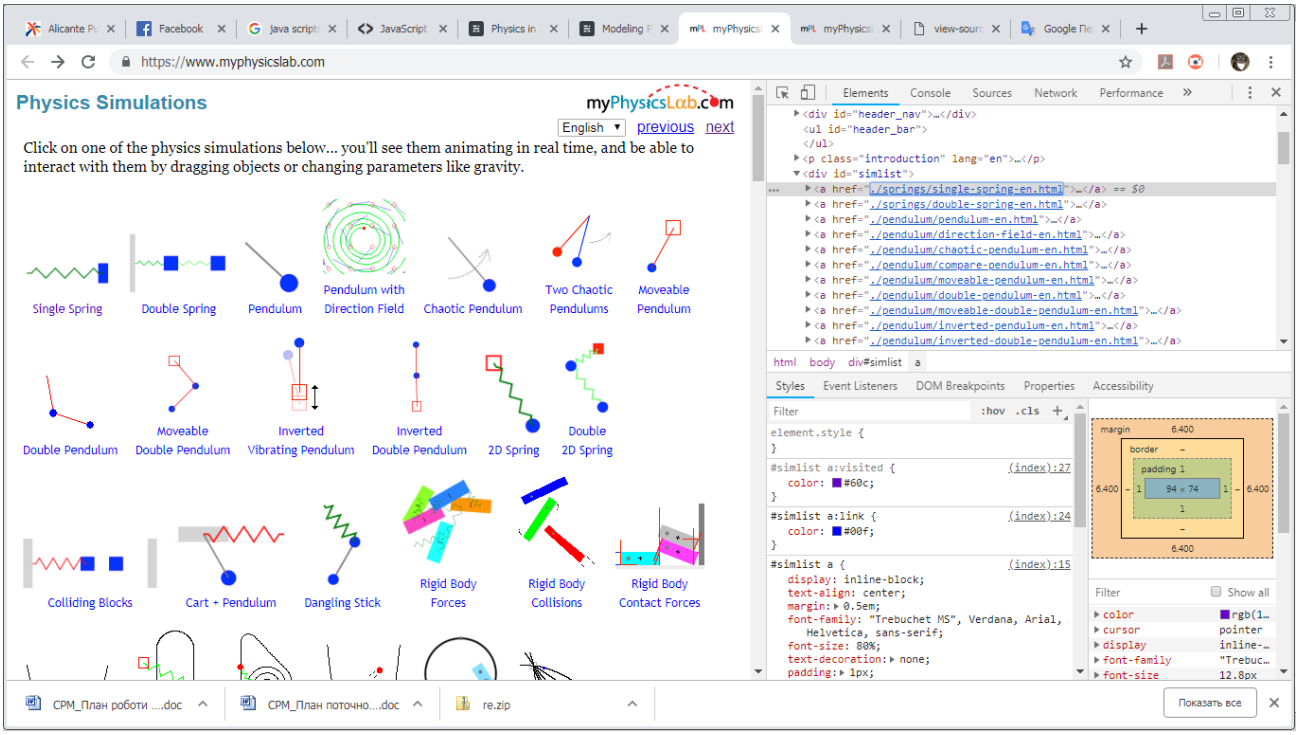
https://www.skillshare.com/classes/The-Ultimate-Guide-to-JavaScript-Physics/1388244587

Про цей курс занять

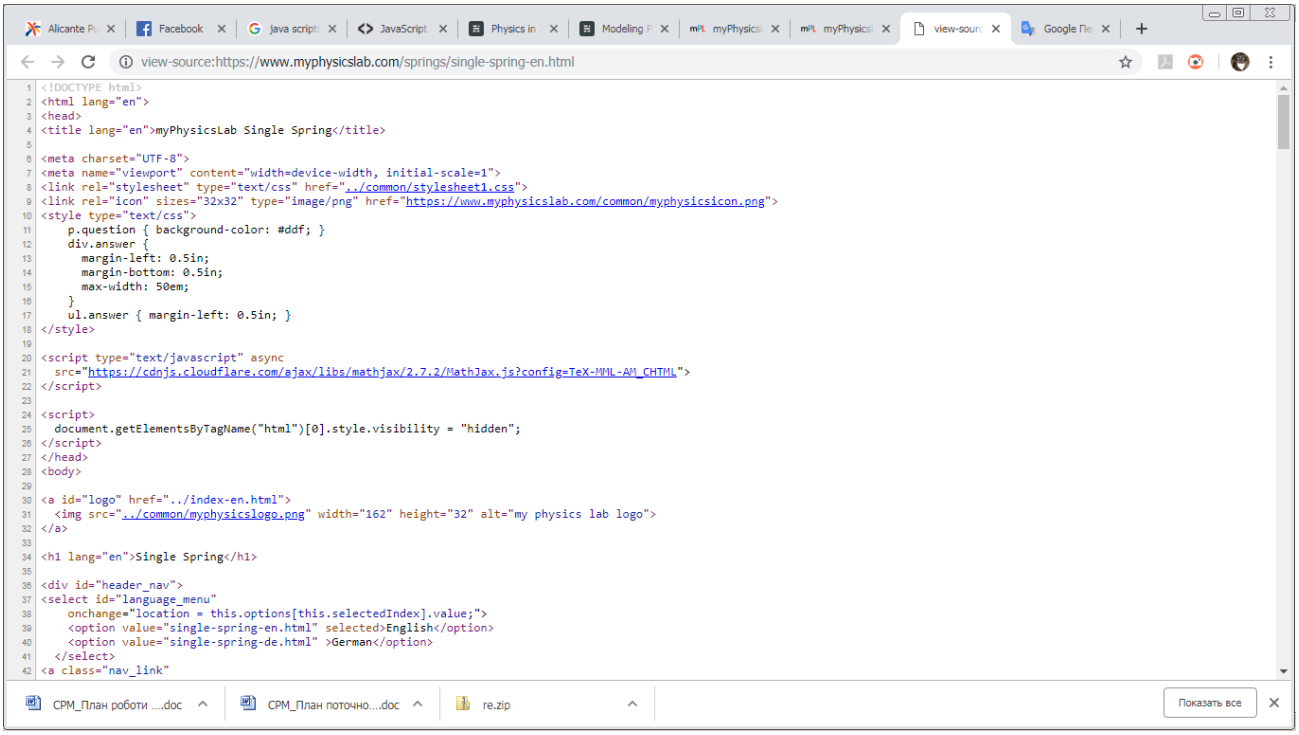
Використання веб-консолі Firefox. Цей курс описує основи використання веб-консолі як калькулятора, як призначати числові змінні та робити основні обчислення.
Що ви дізнаєтеся в цьому курсі:
- Як JavaScript займається порядком операцій (PEMDAS) у веб-консолі Firefox і як уникнути основних помилок обчислення.
- Як об'єкт Math працює в JavaScript і дає приклади його використання в веб-консолі firefox.
- У цьому навчальному посібнику представлено scratchpad Mozilla - спосіб виконання і збереження скриптів.
- Як налаштувати та вирішити основну проблему балансу сил, що включає тригонометрію. Scratchpad викликається для виконання розрахунків.
- Як продемонструвати альтернативний спосіб виконання проблеми в частині 1 і ввести функцію JavaScript atan2.
- Як знайти діапазон пакета, що падає з літака, що рухається з постійною швидкістю з нахилом по відношенню до горизонталі. Консоль javascript використовується для виконання розрахунків.
- Як оцінювати швидкість і прискорення з урахуванням амплітуди руху і кутової швидкості - кутова частота гармонічного руху. Це трохи швидко і брудно і, ймовірно, потрібно переробити для повного суспільного споживання.
- Як амплітуда руху в системах з простим гармонійним рухом пов'язана з тангенціальною швидкістю системи.
- Як кутова частота пов'язана з масою і сталою пружності пружини. Показано, як використовувати джерело для вимірювання інерції об'єкта. Це призначено як фонове відео: ніяких обчислень за допомогою javascript не зроблено.
- Як математика за маятниковим рухом пов'язана з математикою в системі мас струн. Зверніть увагу, що помилка в обчисленні крутного моменту зроблена, зафіксована і виправлена. Це призначено як фонове відео: ніяких обчислень за допомогою javascript не зроблено.
- Як використовувати інструмент аналізу OpenTrack для візуалізації синусоїд і як змінюється амплітуда, кутова частота і фаза зміни форми хвилі, яка з'являється. Суперпозиція також покрита. Javascript "для" циклів і масивів також вводяться.

2. Physics Simulations
https://www.myphysicslab.com/

Приклад задач



Код сторинки



3. Modeling Physics in Javascript: Gravity and Drag

https://burakkanber.com/blog/modeling-physics-javascript-gravity-and-drag/

This is the first post in the Modeling Physics in Javascript series. As such, we'll need to cover some introductory material. There are some of you who have real training in physics and calculus. You may be tempted to scream "that's not the whole story" at me when I explain some concepts, and my response to you is this: I wish I could cover all the amazing parts of physics and calculus in one blog post, but I can't. I'll therefore cover only the bits I need. Hopefully, the people reading this will accidentally learn physics while trying to pick up some cool JS tricks.

Yes, there's a JSFiddle to play with at the bottom of the page!

The Problem

Model a bouncy ball, under the influence of gravity, which can bounce off of horizontal or vertical walls. Oh, and it experiences air drag.

Not too shabby for day 1! We'll first talk about Newton's 2nd Law, forces, gravity, and a touch of calculus before we can start coding.

Newton's 2nd Law

Newton's 2nd Law of Motion basically is physics. It describes how forces interact with macroscopic objects. If you've ever seen a ball fly through the air, or a car moving, or something rolling down a hill, or if you've ever felt anything, you know (indirectly) about Newton's 2nd Law.

Newton's 2nd Law looks like this: F = ma.

Well, not quite. It looks more like this: Fnet = ma.

As you can imagine, physics has a manner of starting simple and then getting very complicated very quickly. The subscript "net" means that the "F" doesn't just refer to a single force, but the sum total of all forces on the object. The bold "F" and "a" mean that those variables (force and acceleration) are vectors -- meaning that they exist separately in 3 different directions (X, Y, and Z).

But that's not too bad. If you don't want to do vector math, you can simply rewrite Fnet = ma as the following three separate equations:

$$Fx, net = max$$
$$Fy, net = may$$
$$Fz, net = maz$$

You may think that having three equations to worry about makes things tougher, but trust me, it doesn't. Just remember that you have to do the F = ma equation for each direction. We'll very rarely use the Z direction, so all we really have to worry about stuff happening in the X and Y directions (left/right, and up/down).

Let's just jump into a simple example. You have a ball with a mass of 2 kg, and there are three forces acting on it:

2 Newtons pulling right
4 N pulling left
1 N pulling down

We combine the left and right forces to get our overall Fx, net = -2 N. (4 to the left and 2 to the right leaves us with 2 left over pulling to the left. The number is negative because in general, we'll consider "right" to be positive and "left" to be negative.)

And since there's only one force acting in the Y direction, the Fy, net is just -1 N. (Let's say "up" is positive and "down" is negative -- but only for now. We'll switch that up later. In general, you use positive and negative signs in a way that makes sense for your specific problem.)

Finally, since our ball weighs 2 kg, we can substitute our numbers into F = ma and get the following two equations:

$$-2\ N = 2\ kg * ax$$
$$-1\ N = 2\ kg * ay$$

So far, so good. Truth be told, we're really interested in the accelerations here, so we'll just rearrange:

$$ax = -2\ N / 2\ kg$$
$$ay = -1\ N / 2\ kg$$

And then I'll tell you that when you divide "Newtons" by "kilograms" you get "meters per second-squared":

$$ax = -1\ m / s2$$
$$ay = -0.5\ m / s2$$

So we had a situation where we know the mass of an object and the forces acting on it. We added up the forces (assigning negative signs to "left" and "down", causing those forces to subtract instead), and rearranged to solve for acceleration -- and we did this for both the X and the Y directions separately. In this example, the ball is accelerating down and to the left, but it's accelerating more to the left than it is downwards.

(More advanced: sometimes you don't know the X and Y components of a force. It's possible that you only know about one force that's angled at, say, 30 degrees up from right. In that case, you can use cosine and sine to figure out the X and Y pieces of that force respectively. We'll do that in a future article.)

Gravity

Gravity is one of the universal fundamental interactions, along with the electromagnetic, strong, and weak forces. Gravity causes a force to act on every single object from every other single object, but fortunately everything works out to simplify nicely if you're on the surface of a planet. On Earth, for instance, we don't even need to worry about the force that gravity causes, we can just look at the end result: every single object on Earth experiences an additional downward acceleration of 9.8 m / s2. That simple.

If our ball from above (that we already solved for) is on Earth and experiencing gravity, then we just need to modify the ay equation slightly. We'll take the downward acceleration that the force is causing and combine it with the downward acceleration that gravity causes:

$$ay = -0.5\ m / s2 - 9.8\ m / s2$$

That leaves us with an acceleration of

$$ay = -10.3\ m / s2$$

As you can see, the downward force combined with the acceleration from gravity shoots the ball downward even faster than either gravity or the force alone could.

So, gravity is simple (if you're on a planet). All you do is modify the ayacceleration to factor in the downward pull of gravity.

In a future article we'll look at how to model gravity if you're not on a planet (perhaps you are a planet), but that doesn't come until later...

Aerodynamic Drag

Our ball experiences gravity, but as I mentioned, we don't need to model that as a force; we just plug it directly into the acceleration result. Drag, however, is a force, and we'll have to model it.

The equation (one of the equations) for aerodynamic drag looks like this:

$$FD = -0.5 * CD * A * \rho * v2$$

It looks complicated, but when we break it down, it's pretty simple. First off, notice the bold characters. Like F = ma above, a bold variable means it's actually a vector -- so right off the bat we can split this into two equations:

$$FD, x = -0.5 * CD * A * \rho * vx2$$
$$FD, y = -0.5 * CD * A * \rho * vy2$$

And then we look at each one of those terms above:

CD is the "coefficient of drag", which is influenced by the shape of the object (and a little bit by its material). For a ball, this is 0.47, and is a dimensionless quantity.

A is the frontal area or frontal projection of the object. If you look at a silhouette of the object from the front, this is the area of that shape. For a ball, the frontal area is just the area of a circle, or π r2.

ρ (Greek letter rho) is the density of the fluid the ball is in. If our ball's in air, this value is 1.22 (kg / m3)

Velocity squared -- since we're looking at this in two directions separately, we use the X velocity and the Y velocity respectively.

Note the -0.5 at the beginning. The negative sign, with the fact that the equation uses velocity, indicates that this force pushes in the opposite direction the ball is moving at all times. Because the velocity is squared it'll always be positive, which means the whole equation will always be negative, ie, opposite the velocity.

A Touch of Calculus

Calculus, like physics, is amazing and has a wonderful depth that I can't do justice in a blog post.

The "derivative" in calculus describes how something changes as something else is changing; often this will be called the "rate of change". When you drive a car at 30 MPH, your position is changing by 30 miles every hour. Your position changes as time changes. It can then be said that velocity is the "derivative of position with respect to time" or simply the "time derivative of position".

Then we can think about what happens when you speed up or slow down (accelerate). You might change your velocity by 5 MPH per hour (MPHPH?). In that sense, your velocity is changing with time, and you can say that acceleration is the time derivative of velocity.

So it starts with position. The derivative of position is velocity. And the derivative of velocity is acceleration. (The derivative of acceleration is called "jerk", and the derivative of jerk is called "jounce".)

Why is this relevant? Because if you know the acceleration of something (5 meters per second per second), and if you know how fast it's going when you start looking at it (let's say it's not moving at all), you can figure out where it will be at every moment in the future.

An example: your ball is accelerating at 2 m / s2 (meters per second per second). Let's say it's not moving at all when you start looking at it, and that the starting point is called x = 0;

Time	Accel	Velocity	Position
0 s	2 m / s2	0 m / s	0 m
1 s	2 m / s2	2 m / s	0 m
2 s	2 m / s2	4 m / s	2 m
3 s	2 m / s2	6 m / s	6 m
4 s	2 m / s2	8 m / s	12 m
5 s	2 m / s2	10 m / s	20 m
6 s	2 m / s2	12 m / s	30 m

And so on. This is the approach we'll use when solving for the motion of our ball, except we'll do it not once per second but 40 times per second. All we're doing is using our knowledge of the forces to figure out the acceleration at every frame. Then we use the acceleration and current velocity to figure out the new velocity. And then we use the velocity and last position to find the current position.

Writing the Code

Time to dive in. I won't reproduce all the code in snippets, because there's some stuff that has nothing to do with physics. You'll be able to see the full script at the bottom of the page in the JSFiddle.

```
var frameRate = 1/40; // Seconds
var frameDelay = frameRate * 1000; // ms
var loopTimer = false;
var ball = {
        position: {x: width/2, y: 0},
        velocity: {x: 10, y: 0},
        mass: 0.1, //kg
```

```
        radius: 15, // 1px = 1cm
        restitution: -0.7
};
var Cd = 0.47; // Dimensionless
var rho = 1.22; // kg / m^3
var A = Math.PI * ball.radius * ball.radius / (10000);
var ag = 9.81;
```

We set up the frame rate and plug in some physics values. We also create a ball object that stores the ball's position, velocity, mass, radius, and a number called "restitution". You'll see later that this value determines how bouncy the ball is.

Notice here that we've set the ball to be moving at the start of the simulation.

The best part of programming real physics is the fact that you can look up the density of water and replace the value for rho, and the ball will actually behave as if it's in water! If you program the physics correctly, then all you have to do is change the constants and the rest follows. Want the ball to be on the moon? Just change the acceleration due to gravity. Want the ball to swim through water? Just change the density rho. Want a light, floaty beach ball? Lower the mass and increase the radius.

Please play with these values in the JSFiddle below. Change rho and the radius and the mass, and see how physics affects the simulation!

```
var setup = function() {
        canvas = document.getElementById("canvas");
        ctx = canvas.getContext("2d");
        canvas.onmousemove = getMousePosition;
        canvas.onmousedown = mouseDown;
        canvas.onmouseup = mouseUp;
        ctx.fillStyle = 'red';
        ctx.strokeStyle = '#000000';
        loopTimer = setInterval(loop, frameDelay);
}
```

The setup function initializes the canvas and sets up a loop that executes every frameDelay milliseconds. We'll do all the physics and animation in the loop function.

In the loop:

```
// Do physics
// Drag force: Fd = -1/2 * Cd * A * rho * v * v
var Fx = -0.5 * Cd * A * rho * ball.velocity.x * ball.velocity.x * ball.velocity.x / Math.abs(ball.velocity.x);
var Fy = -0.5 * Cd * A * rho * ball.velocity.y * ball.velocity.y * ball.velocity.y / Math.abs(ball.velocity.y);
Fx = (isNaN(Fx) ? 0 : Fx);
Fy = (isNaN(Fy) ? 0 : Fy);
// Calculate acceleration ( F = ma )
var ax = Fx / ball.mass;
var ay = ag + (Fy / ball.mass);
// Integrate to get velocity
ball.velocity.x += ax*frameRate;
ball.velocity.y += ay*frameRate;
// Integrate to get position
ball.position.x += ball.velocity.x*frameRate*100;
ball.position.y += ball.velocity.y*frameRate*100;
```

First off, we calculate the drag forces on the ball. There's a little trick I used to get the direction of the velocity. Instead of using "if" statements to see if the velocity is positive or negative, I just do:

```
ball.velocity.y / Math.abs(ball.velocity.y)
```

at the end of the drag force statements. Dividing a number by its absolute value just leaves the sign. Other than that, the drag force lines are pretty straightforward. We're just calculating the forces.
After that, we calculate acceleration. Notice that the statement for "ay" is different from "ax". Gravity only works in the Y direction, so we add that in here. Also notice that in this problem, "downwards" is positive, unlike the example at the top of the page.
After that, we update the ball velocities with the acceleration times the frame rate. The reason we multiply by the frame rate is so: the acceleration is given in "meters per second-squared". But we're calling this loop 40 times a second (not once a second), so we need to divide by 40 (or multiply by 1/40 in this case) to adjust for the frame rate.
Finally, update the ball positions in a similar fashion. In this case we're also multiplying by 100. If you look at the ball object definition way above you'll see I commented that "1px = 1cm", so this *100 is just an adjustment to make everything work out in meters.
Then we handle collisions with the walls:

```
// Handle collisions
if (ball.position.y > height - ball.radius) {
	ball.velocity.y *= ball.restitution;
	ball.position.y = height - ball.radius;
}
if (ball.position.x > width - ball.radius) {
	ball.velocity.x *= ball.restitution;
	ball.position.x = width - ball.radius;
}
if (ball.position.x < ball.radius) {
	ball.velocity.x *= ball.restitution;
	ball.position.x = ball.radius;
}
```

We're just checking to see if the ball has ended up past the wall in this frame. If it has, then we multiply the velocity in that direction by the restitution coefficient from above. Since that number is always negative, it'll make the ball reverse direction. If you set the restitution to -1, it'll be perfectly bouncy, meaning it'll bounce up as high as it started falling from. If you set the restitution to 0, it'll flop dead on the ground with no bounce whatsoever. And if you set it to something like -2, it'll bounce even higher than it started. Play with it!
We also modify the position of the ball to just kiss the wall -- this way the ball won't get stuck "in" the wall. Keep in mind that the ball is moving in discrete motions, and so when it collides with the wall it's actually overlapping slightly.
Finally, since we want to be able to control the ball with the mouse, we'll add some handlers (not all code shown here):

```
var mouseDown = function(e) {
	if (e.which == 1) {
		getMousePosition(e);
		mouse.isDown = true;
		ball.position.x = mouse.x;
		ball.position.y = mouse.y;
	}
}
var mouseUp = function(e) {
	if (e.which == 1) {
		mouse.isDown = false;
		ball.velocity.y = (ball.position.y - mouse.y) / 10;
		ball.velocity.x = (ball.position.x - mouse.x) / 10;
	}
}
```

If you click and drag the mouse, and let go, you'll create a kind of slingshot effect. This is achieved not by applying a force to the ball (which you could do), but rather by overriding the velocity of the ball based on how far you pulled the mouse. I like this approach better because it's easier to apply than a force. If you were to use a force to move the ball with the mouse, you'd have to apply the force over a period of time. The "initial velocity" technique above just lets you un-naturally override the velocity in an instant and let physics figure everything out again.

I hope you learned something! Please fork and play with the code in the fiddle below. Click and drag the mouse to slingshot the ball.

Note that certain configurations of variables will cause the simulation to become unstable. Try setting rho = 1000 but leaving the mass at 0.1. The ball should spaz out and blink around the screen. This isn't a problem with the physics, it's just that we're not running at a high enough frame rate for the very large drag forces. To solve that problem we would have to increase the frame rate to make the simulation stable again. Or we could use a different solver (we're using Euler's method here, a first-degree ODE solver) -- but we won't talk about solvers for a few weeks.

```
/* Burak Kanber */
var width = 500;
var height = 400;
var canvas = ctx = false;
var frameRate = 1/40; // Seconds
var frameDelay = frameRate * 1000; // ms
var loopTimer = false;
/*
 * Experiment with values of mass, radius, restitution,
 * gravity (ag), and density (rho)!
 *
 * Changing the constants literally changes the environment
 * the ball is in.
 *
 * Some settings to try:
 * the moon: ag = 1.6
 * water: rho = 1000, mass 5
 * beach ball: mass 0.05, radius 30
 * lead ball: mass 10, restitution -0.05
 */
var ball = {
    position: {x: width/2, y: 0},
    velocity: {x: 10, y: 0},
    mass: 0.1, //kg
    radius: 15, // 1px = 1cm
    restitution: -0.7
    };
var Cd = 0.47;  // Dimensionless
var rho = 1.22; // kg / m^3
var A = Math.PI * ball.radius * ball.radius / (10000); // m^2
var ag = 9.81;  // m / s^2
var mouse = {x: 0, y: 0, isDown: false};
function getMousePosition(e) {
    mouse.x = e.pageX - canvas.offsetLeft;
    mouse.y = e.pageY - canvas.offsetTop;
}
var mouseDown = function(e) {
    if (e.which == 1) {
        getMousePosition(e);
        mouse.isDown = true;
        ball.position.x = mouse.x;
        ball.position.y = mouse.y;
    }
}
var mouseUp = function(e) {
    if (e.which == 1) {
        mouse.isDown = false;
        ball.velocity.y = (ball.position.y - mouse.y) /10;
        ball.velocity.x = (ball.position.x - mouse.x) / 10;
    }
}
var setup = function() {
    canvas = document.getElementById("canvas");
    ctx = canvas.getContext("2d");

    canvas.onmousemove = getMousePosition;
    canvas.onmousedown = mouseDown;
    canvas.onmouseup = mouseUp;
```

```
        ctx.fillStyle = 'red';
    ctx.strokeStyle = '#000000';
    loopTimer = setInterval(loop, frameDelay);
}
var loop = function() {
    if ( ! mouse.isDown) {
        // Do physics
            // Drag force: Fd = -1/2 * Cd * A * rho * v * v
        var Fx = -0.5 * Cd * A * rho * ball.velocity.x * ball.velocity.x * ball.velocity.x / Math.abs(ball.velocity.x);
        var Fy = -0.5 * Cd * A * rho * ball.velocity.y * ball.velocity.y * ball.velocity.y / Math.abs(ball.velocity.y);

        Fx = (isNaN(Fx) ? 0 : Fx);
        Fy = (isNaN(Fy) ? 0 : Fy);
                    // Calculate acceleration ( F = ma )
        var ax = Fx / ball.mass;
        var ay = ag + (Fy / ball.mass);
            // Integrate to get velocity
        ball.velocity.x += ax*frameRate;
        ball.velocity.y += ay*frameRate;

            // Integrate to get position
        ball.position.x += ball.velocity.x*frameRate*100;
        ball.position.y += ball.velocity.y*frameRate*100;
    }
    // Handle collisions
    if (ball.position.y > height - ball.radius) {
        ball.velocity.y *= ball.restitution;
        ball.position.y = height - ball.radius;
    }
    if (ball.position.x > width - ball.radius) {
        ball.velocity.x *= ball.restitution;
        ball.position.x = width - ball.radius;
    }
    if (ball.position.x < ball.radius) {
        ball.velocity.x *= ball.restitution;
        ball.position.x = ball.radius;
    }
    // Draw the ball
    ctx.clearRect(0,0,width,height);
    ctx.save();
    ctx.translate(ball.position.x, ball.position.y);
    ctx.beginPath();
    ctx.arc(0, 0, ball.radius, 0, Math.PI*2, true);
    ctx.fill();
    ctx.closePath();

    ctx.restore();
    // Draw the slingshot
    if (mouse.isDown) {
        ctx.beginPath();
        ctx.moveTo(ball.position.x, ball.position.y);
        ctx.lineTo(mouse.x, mouse.y);
        ctx.stroke();
        ctx.closePath();
    }
    }
    setup();
```

4. Physics in Javascript: Rigid Bodies -- Part 1 (Pendulum Clock)

https://burakkanber.com/blog/physics-in-javascript-rigid-bodies-part-1-pendulum-clock/

5. JavaScript Physics with Matter.js

https://codersblock.com/blog/javascript-physics-with-matter-js/