Case Study: $N$-Body Problem
Outline

1. N-Body Simulation
2. Body Data Type
3. Universe Data Type
Newton’s first law of motion states that a body in motion remains in motion at the same velocity unless acted on by an outside force.

Newton’s second law of motion explains how outside forces on a body affect its velocity.

The $N$-body simulation problem, originally formulated by Isaac Newton over 350 years ago, describes the motion of the $N$ bodies, mutually affected by gravitational forces.
**N-Body Simulation**

Newton’s first law of motion states that a body in motion remains in motion at the same velocity unless acted on by an outside force.

Newton’s second law of motion explains how outside forces on a body affect its velocity.

The $N$-body simulation problem, originally formulated by Isaac Newton over 350 years ago, describes the motion of the $N$ bodies, mutually affected by gravitational forces.
Newton’s first law of motion states that a body in motion remains in motion at the same velocity unless acted on by an outside force.

Newton’s second law of motion explains how outside forces on a body affect its velocity.

The $N$-body simulation problem, originally formulated by Isaac Newton over 350 years ago, describes the motion of the $N$ bodies, mutually affected by gravitational forces.
Newton’s first law of motion states that a body in motion remains in motion at the same velocity unless acted on by an outside force.

Newton’s second law of motion explains how outside forces on a body affect its velocity.

The $N$-body simulation problem, originally formulated by Isaac Newton over 350 years ago, describes the motion of the $N$ bodies, mutually affected by gravitational forces.
**Body Data Type**

A data type `Body` for moving bodies

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Body(r, v, mass)</td>
<td>a new body (b) with mass (mass) at position (r) moving at velocity (v)</td>
</tr>
<tr>
<td>b.move(f, dt)</td>
<td>move (b) by applying force (f) for (dt) seconds</td>
</tr>
<tr>
<td>b.forceFrom(a)</td>
<td>force vector from body (a) on (b)</td>
</tr>
<tr>
<td>b.draw()</td>
<td>draw (b) to standard draw</td>
</tr>
</tbody>
</table>

Note that `Body` is a `Vector` client — the values of the data type are `Vector` objects that carry the body’s position and velocity, as well as a float that carries the mass.

We use Newton’s second law \((\mathbf{F} = ma)\) for updating the position and velocity of a body due to a given force vector \(\mathbf{f}\) and amount of time \(dt\):

\[
a = f.\text{scale}(1.0 / \text{mass})
\]
\[
v = v + a.\text{scale}(dt)
\]
\[
r = r + v.\text{scale}(dt)
\]
## Body Data Type

A data type `Body` for moving bodies

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Body(r, v, mass)</code></td>
<td>a new body <code>b</code> with mass <code>mass</code> at position <code>r</code> moving at velocity <code>v</code></td>
</tr>
<tr>
<td><code>b.move(f, dt)</code></td>
<td>move <code>b</code> by applying force <code>f</code> for <code>dt</code> seconds</td>
</tr>
<tr>
<td><code>b.forceFrom(a)</code></td>
<td>force vector from body <code>a</code> on <code>b</code></td>
</tr>
<tr>
<td><code>b.draw()</code></td>
<td>draw <code>b</code> to standard draw</td>
</tr>
</tbody>
</table>

Note that `Body` is a `Vector` client — the values of the data type are `Vector` objects that carry the body’s position and velocity, as well as a float that carries the mass.

We use Newton’s second law ($F = ma$) for updating the position and velocity of a body due to a given force vector $f$ and amount of time $dt$:

\[
a = f \cdot \text{scale}(1.0 / \text{mass})
\]

\[
v = v + a \cdot \text{scale}(dt)
\]

\[
r = r + v \cdot \text{scale}(dt)
\]
## Body Data Type

A data type `Body` for moving bodies

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Body(r, v, mass)</code></td>
<td>a new body <code>b</code> with mass <code>mass</code> at position <code>r</code> moving at velocity <code>v</code></td>
</tr>
<tr>
<td><code>b.move(f, dt)</code></td>
<td>move <code>b</code> by applying force <code>f</code> for <code>dt</code> seconds</td>
</tr>
<tr>
<td><code>b.forceFrom(a)</code></td>
<td>force vector from body <code>a</code> on <code>b</code></td>
</tr>
<tr>
<td><code>b.draw()</code></td>
<td>draw <code>b</code> to standard draw</td>
</tr>
</tbody>
</table>

Note that `Body` is a `Vector` client — the values of the data type are `Vector` objects that carry the body’s position and velocity, as well as a float that carries the mass.

We use Newton’s second law ($F = ma$) for updating the position and velocity of a body due to a given force vector $f$ and amount of time $dt$:

\[
a = f \cdot \text{scale}(1.0 / \text{mass})
\]

\[
v = v + a \cdot \text{scale}(dt)
\]

\[
r = r + v \cdot \text{scale}(dt)
\]
Body Data Type

A data type `Body` for moving bodies

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Body(r, v, mass)</code></td>
<td>a new body <code>b</code> with mass <code>mass</code> at position <code>r</code> moving at velocity <code>v</code></td>
</tr>
<tr>
<td><code>b.move(f, dt)</code></td>
<td>move <code>b</code> by applying force <code>f</code> for <code>dt</code> seconds</td>
</tr>
<tr>
<td><code>b.forceFrom(a)</code></td>
<td>force vector from body <code>a</code> on <code>b</code></td>
</tr>
<tr>
<td><code>b.draw()</code></td>
<td>draw <code>b</code> to standard draw</td>
</tr>
</tbody>
</table>

Note that `Body` is a `Vector` client — the values of the data type are `Vector` objects that carry the body’s position and velocity, as well as a float that carries the mass.

We use Newton’s second law \((F = ma)\) for updating the position and velocity of a body due to a given force vector \(\mathbf{f}\) and amount of time \(dt\):

\[
\begin{align*}
a &= \mathbf{f}.\text{scale}(1.0 / \text{mass}) \\
v &= v + a.\text{scale}(dt) \\
r &= r + v.\text{scale}(dt)
\end{align*}
\]
We use Newton’s law of universal gravitation \( F = -G \frac{m_1 m_2}{r^2} \hat{r} \) for computing the force imposed on one body by another.

\[
G = 6.67 \times 10^{-11} \\
delta = b._r - a._r \\
dist = abs(delta) \\
magnitude = G * a._mass * b._mass / (dist * dist) \\
f = delta.direction().scale(magnitude)
\]
We use Newton’s law of universal gravitation \( \mathbf{F} = -G \frac{m_1 m_2}{r^2} \hat{r} \) for computing the force imposed on one body by another.

\[
G = 6.67 \times 10^{-11} \\
delta = b._r - a._r \\
dist = \text{abs}(\delta) \\
magnitude = G \times a._\text{mass} \times b._\text{mass} / (\text{dist} \times \text{dist}) \\
f = \delta\text{.direction()}.\text{scale(magnitude)}
\]
import stddraw

class Body:
    def __init__(self, r, v, mass):
        self._r = r
        self._v = v
        self._mass = mass

    def move(self, f, dt):
        a = f.scale(1 / self._mass)
        self._v = self._v + (a.scale(dt))
        self._r = self._r + self._v.scale(dt)

    def forceFrom(self, other):
        G = 6.67e-11
        delta = other._r - self._r
        dist = abs(delta)
        magnitude = (G * self._mass * other._mass) / (dist * dist)
        return delta.direction().scale(magnitude)

    def draw(self):
        stddraw.setPenRadius(0.0125)
        stddraw.point(self._r[0], self._r[1])
Body Data Type

body.py: Definition of Body data type representing a gravitating body.

```python
import stddraw

class Body:
    def __init__(self, r, v, mass):
        self._r = r
        self._v = v
        self._mass = mass

    def move(self, f, dt):
        a = f.scale(1 / self._mass)
        self._v = self._v + (a.scale(dt))
        self._r = self._r + self._v.scale(dt)

    def forceFrom(self, other):
        G = 6.67e-11
        delta = other._r - self._r
        dist = abs(delta)
        magnitude = (G * self._mass * other._mass) / (dist * dist)
        return delta.direction().scale(magnitude)

    def draw(self):
        stddraw.setPenRadius(0.0125)
        stddraw.point(self._r[0], self._r[1])
```

Universe Data Type

The data type Universe models the universe

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universe(file)</td>
<td>a new universe ( u ) built from a description in file</td>
</tr>
<tr>
<td>( u.\text{increaseTime}(dt) )</td>
<td>update ( u ) by simulating the universe for ( dt ) seconds</td>
</tr>
<tr>
<td>( u.\text{draw()} )</td>
<td>draw universe ( u ) to standard draw</td>
</tr>
</tbody>
</table>

The constructor reads the universe parameters and body descriptions from a file that contains the following information

- The number of bodies
- The radius of the universe
- The position, velocity, and mass of each body

Example (parameters for a 2-body system)

```
$ more 2body.txt
2
5.0e10
0.0e00 4.5e10 1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30
```
Universe Data Type

The data type `Universe` models the universe

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Universe(file)</code></td>
<td>a new universe ( u ) built from a description in ( file )</td>
</tr>
<tr>
<td><code>u.increaseTime(dt)</code></td>
<td>update ( u ) by simulating the universe for ( dt ) seconds</td>
</tr>
<tr>
<td><code>u.draw()</code></td>
<td>draw universe ( u ) to standard draw</td>
</tr>
</tbody>
</table>

The constructor reads the universe parameters and body descriptions from a file that contains the following information

- The number of bodies
- The radius of the universe
- The position, velocity, and mass of each body

Example (parameters for a 2-body system)

```
$ more 2body.txt
2
5.0e10
0.0e00 4.5e10 1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30
```
Universe Data Type

The data type `Universe` models the universe

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Universe(file)</code></td>
<td>a new universe <code>u</code> built from a description in <code>file</code></td>
</tr>
<tr>
<td><code>u.increaseTime(dt)</code></td>
<td>update <code>u</code> by simulating the universe for <code>dt</code> seconds</td>
</tr>
<tr>
<td><code>u.draw()</code></td>
<td>draw universe <code>u</code> to standard draw</td>
</tr>
</tbody>
</table>

The constructor reads the universe parameters and body descriptions from a file that contains the following information

- The number of bodies
- The radius of the universe
- The position, velocity, and mass of each body

Example (parameters for a 2-body system)

```
$ more 2body.txt
2
5.0e10
0.0e00 4.5e10 1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30
```
The data type `Universe` models the universe

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Universe(file)</code></td>
<td>a new universe $u$ built from a description in $file$</td>
</tr>
<tr>
<td><code>u.increaseTime(dt)</code></td>
<td>update $u$ by simulating the universe for $dt$ seconds</td>
</tr>
<tr>
<td><code>u.draw()</code></td>
<td>draw universe $u$ to standard draw</td>
</tr>
</tbody>
</table>

The constructor reads the universe parameters and body descriptions from a file that contains the following information:

- The number of bodies
- The radius of the universe
- The position, velocity, and mass of each body

Example (parameters for a 2-body system)

```bash
$ more 2body.txt
2
5.0e10
0.0e00 4.5e10 1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30
```
universe.py: Accept a string *filename* and a float *dt* as command-line arguments, and simulate the motion in the universe defined by the contents of *filename*, increasing time at the rate specified by *dt*.

```python
import stdarray
import stddraw
import sys
from body import Body
from instream import InStream
from vector import Vector

class Universe:
    def __init__(self, filename):
        instream = InStream(filename)
        n = instream.readInt()
        radius = instream.readFloat()
        stddraw.setXscale(-radius, +radius)
        stddraw.setYscale(-radius, +radius)
        self._bodies = stdarray.create1D(n)
        for i in range(n):
            rx = instream.readFloat()
            ry = instream.readFloat()
            vx = instream.readFloat()
            vy = instream.readFloat()
            mass = instream.readFloat()
            r = Vector([rx, ry])
            v = Vector([vx, vy])
            self._bodies[i] = Body(r, v, mass)
```
universe.py: Accept a string `filename` and a float `dt` as command-line arguments, and simulate the motion in the universe defined by the contents of `filename`, increasing time at the rate specified by `dt`.

```python
import stdarray
import stddraw
import sys
from body import Body
from instream import InStream
from vector import Vector

class Universe:
    def __init__(self, filename):
        instream = InStream(filename)
        n = instream.readInt()
        radius = instream.readFloat()
        stddraw.setXscale(-radius, +radius)
        stddraw.setYscale(-radius, +radius)
        self._bodies = stdarray.create1D(n)
        for i in range(n):
            rx = instream.readFloat()
            ry = instream.readFloat()
            vx = instream.readFloat()
            vy = instream.readFloat()
            mass = instream.readFloat()
            r = Vector([rx, ry])
            v = Vector([vx, vy])
            self._bodies[i] = Body(r, v, mass)
```
def increaseTime(self, dt):
    n = len(self._bodies)
    f = stdarray.create1D(n, Vector([0, 0]))
    for i in range(n):
        for j in range(n):
            if i != j:
                bodyi = self._bodies[i]
                bodyj = self._bodies[j]
                f[i] = f[i] + bodyi.forceFrom(bodyj)
    for i in range(n):
        self._bodies[i].move(f[i], dt)

def draw(self):
    for body in self._bodies:
        body.draw()

def main():
    filename = sys.argv[1]
    dt = float(sys.argv[2])
    universe = Universe(filename)
    while True:
        universe.increaseTime(dt)
        stddraw.clear()
        universe.draw()
        stddraw.show(10)

if __name__ == '__main__':
    main()
def increaseTime(self, dt):
    n = len(self._bodies)
    f = stdarray.create1D(n, Vector([0, 0]))
    for i in range(n):
        for j in range(n):
            if i != j:
                bodyi = self._bodies[i]
                bodyj = self._bodies[j]
                f[i] = f[i] + bodyi.forceFrom(bodyj)

    for i in range(n):
        self._bodies[i].move(f[i], dt)

def draw(self):
    for body in self._bodies:
        body.draw()

def main():
    filename = sys.argv[1]
    dt = float(sys.argv[2])
    universe = Universe(filename)
    while True:
        universe.increaseTime(dt)
        stddraw.clear()
        universe.draw()
        stddraw.show(10)

if __name__ == '__main__':
    main()
Universe Data Type

A 2-body system

```bash
$ more 2body.txt
2
5.0e10
0.0e00 4.5e10 1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30

$ python universe.py 2body.txt 20000
```

A 3-body system

```bash
$ more 3body.txt
3
1.25e11
0.0e00 0.0e00 0.05e04 0.0e00 5.97e24
0.0e00 4.5e10 3.0e04 0.0e00 1.989e30
0.0e00 -4.5e10 -3.0e04 0.0e00 1.989e30

$ python universe.py 3body.txt 20000
```

A 4-body system

```bash
$ more 4body.txt
4
5.0e10
-3.5e10 0.0e00 0.0e00 1.4e03 3.0e28
-1.0e10 0.0e00 0.0e00 1.4e04 3.0e28
1.0e10 0.0e00 0.0e00 -1.4e04 3.0e28
3.5e10 0.0e00 0.0e00 -1.4e03 3.0e28

$ python universe.py 4body.txt 20000
```
Universe Data Type
A 2-body system

$ more 2body.txt
2
5.0e10
0.0e00 4.5e10 1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30

$ python universe.py 2body.txt 20000

A 3-body system

$ more 3body.txt
3
1.25e11
0.0e00 0.0e00 0.05e04 0.0e00 5.97e24
0.0e00 4.5e10 3.0e04 0.0e00 1.989e30
0.0e00 -4.5e10 -3.0e04 0.0e00 1.989e30

$ python universe.py 3body.txt 20000

A 4-body system

$ more 4body.txt
4
5.0e10
-3.5e10 0.0e00 0.0e00 1.4e03 3.0e28
-1.0e10 0.0e00 0.0e00 1.4e04 3.0e28
1.0e10 0.0e00 0.0e00 -1.4e04 3.0e28
3.5e10 0.0e00 0.0e00 -1.4e03 3.0e28

$ python universe.py 4body.txt 20000
Universe Data Type

A 2-body system

$ more 2body.txt

2
5.0e10
0.0e00 4.5e10 1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30

$ python universe.py 2body.txt 20000

A 3-body system

$ more 3body.txt

3
1.25e11
0.0e00 0.0e00 0.05e04 0.0e00 5.97e24
0.0e00 4.5e10 3.0e04 0.0e00 1.989e30
0.0e00 -4.5e10 -3.0e04 0.0e00 1.989e30

$ python universe.py 3body.txt 20000

A 4-body system

$ more 4body.txt

4
5.0e10
-3.5e10 0.0e00 0.0e00 1.4e03 3.0e28
-1.0e10 0.0e00 0.0e00 1.4e04 3.0e28
1.0e10 0.0e00 0.0e00 -1.4e04 3.0e28
3.5e10 0.0e00 0.0e00 -1.4e03 3.0e28

$ python universe.py 4body.txt 20000
Universe Data Type
A 2-body system

$ more 2body.txt
2
5.0e10
0.0e00 4.5e10 1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30

$ python universe.py 2body.txt 20000

A 3-body system

$ more 3body.txt
3
1.25e11
0.0e00 0.0e00 0.05e04 0.0e00 5.97e24
0.0e00 4.5e10 3.0e04 0.0e00 1.989e30
0.0e00 -4.5e10 -3.0e04 0.0e00 1.989e30

$ python universe.py 3body.txt 20000

A 4-body system

$ more 4body.txt
4
5.0e10
-3.5e10 0.0e00 0.0e00 1.4e03 3.0e28
-1.0e10 0.0e00 0.0e00 1.4e04 3.0e28
1.0e10 0.0e00 0.0e00 -1.4e04 3.0e28
3.5e10 0.0e00 0.0e00 -1.4e03 3.0e28

$ python universe.py 4body.txt 20000