Trajectory of a Spinning Object

Magnus Effect

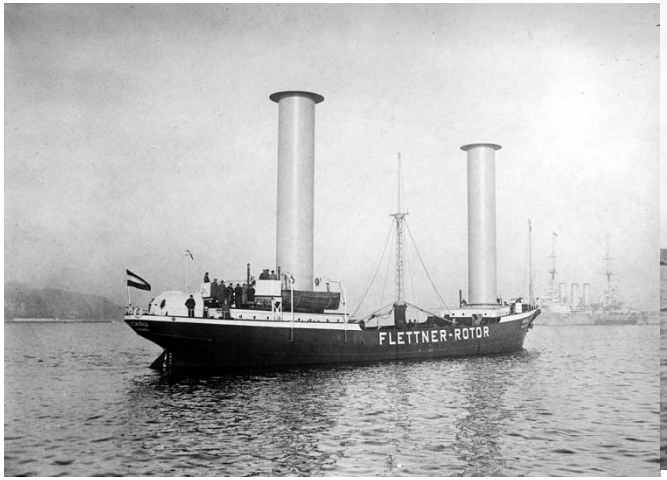
#12

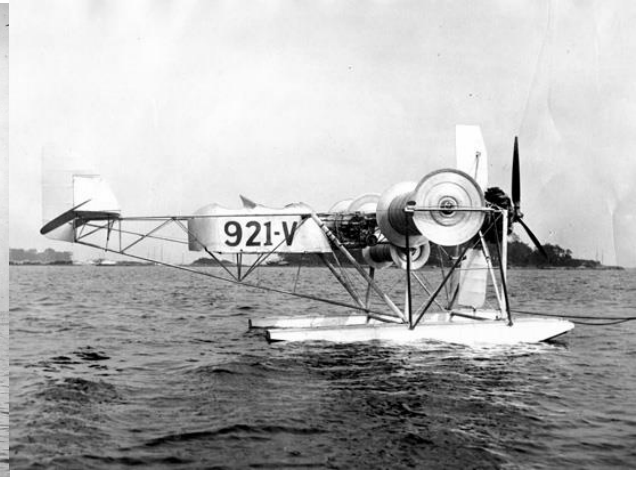
Serdar ARITAN

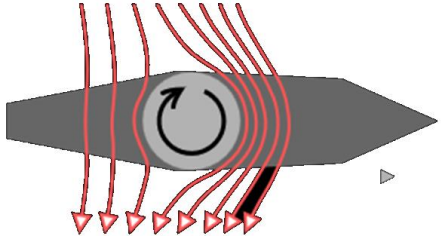
Biomechanics Research Group,
Faculty of Sports Sciences, and
Department of Computer Graphics
Hacettepe University, Ankara, Turkey

Magnus Effect

The Magnus effect is the commonly observed effect in which a spinning ball (or cylinder) curves away from its principal flight path. It is important in many ball sports. It affects spinning missiles, and has some engineering uses, for instance in the design of rotor ships and Flettner aeroplanes.

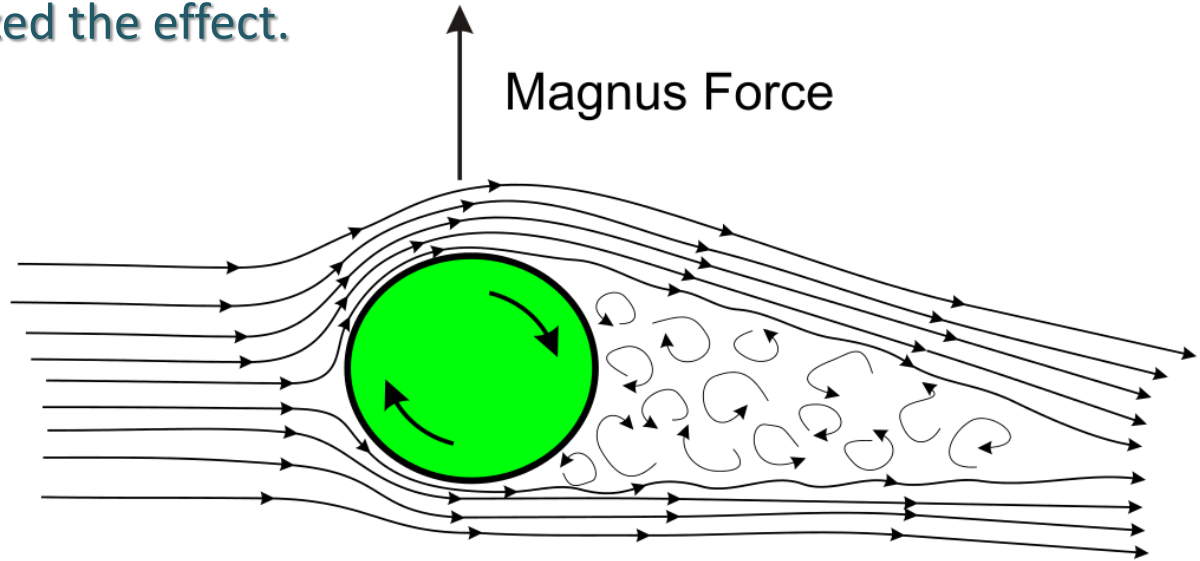




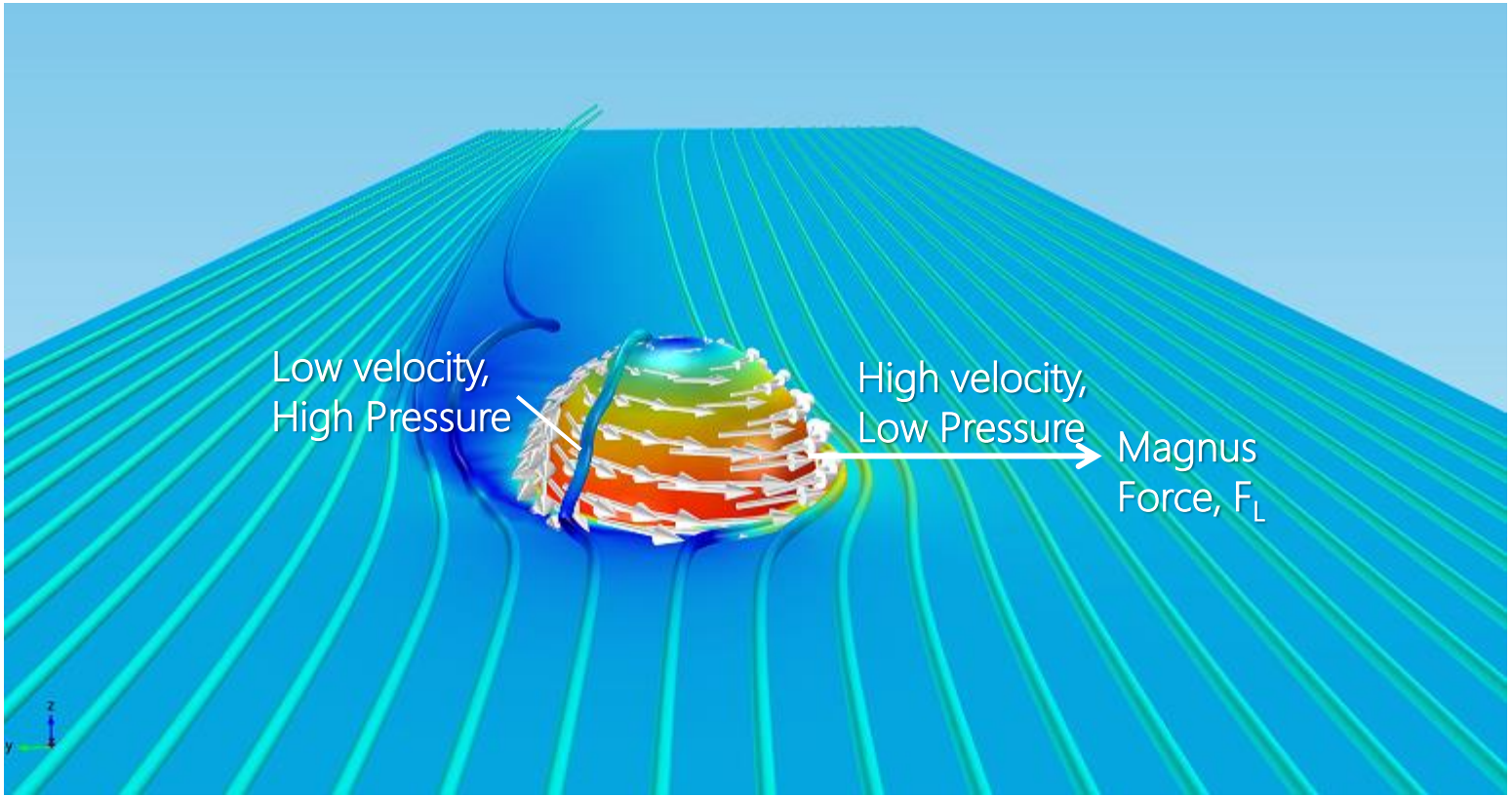


Magnus Effect

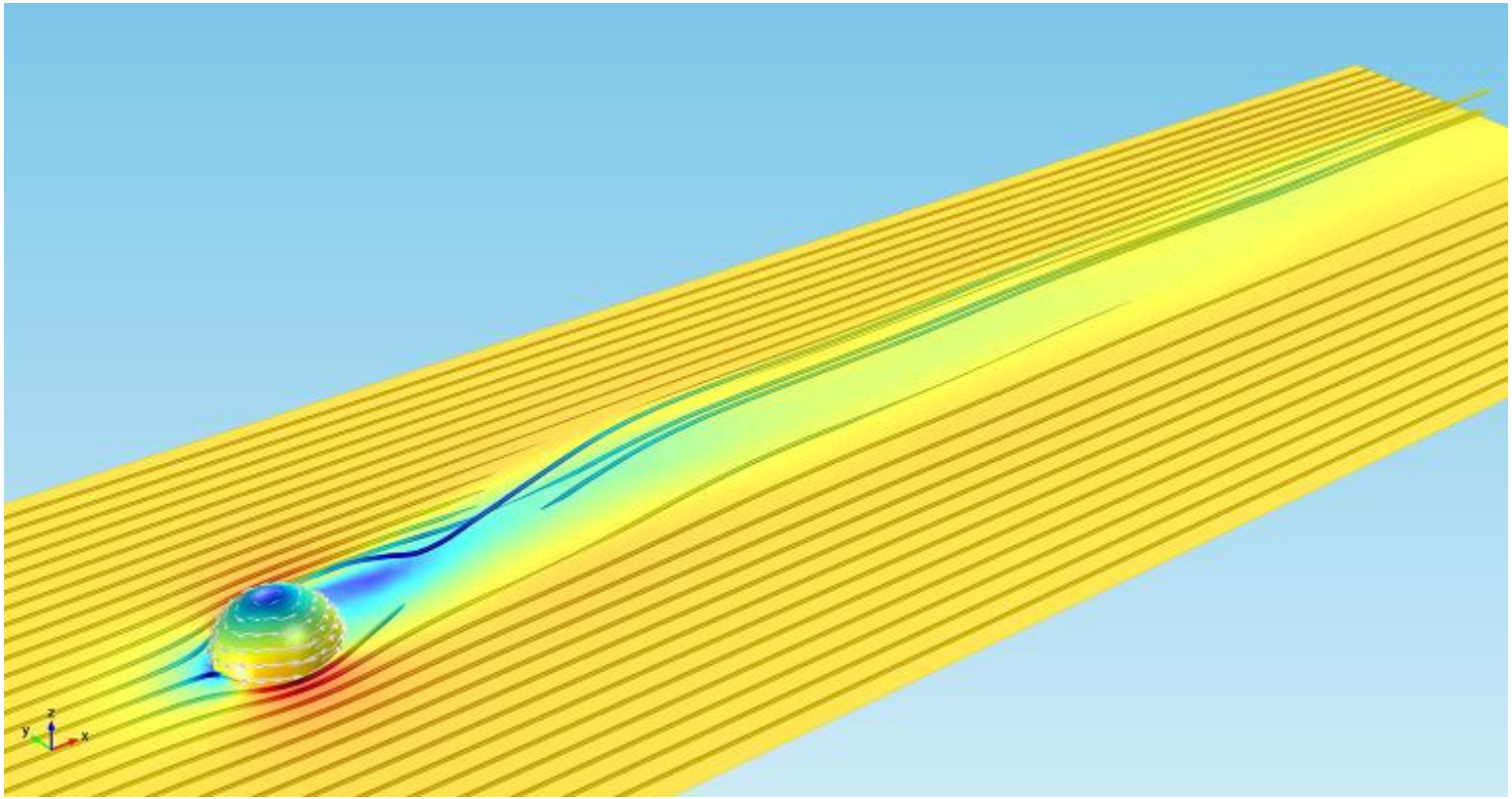
The Magnus effect is named after Gustav Magnus, the German physicist who investigated it. The force on a rotating cylinder is known as **Kutta–Joukowski** lift, after Martin Wilhelm Kutta and Nikolai Zhukovsky (or Joukowski), who first analyzed the effect.



Magnus Effect



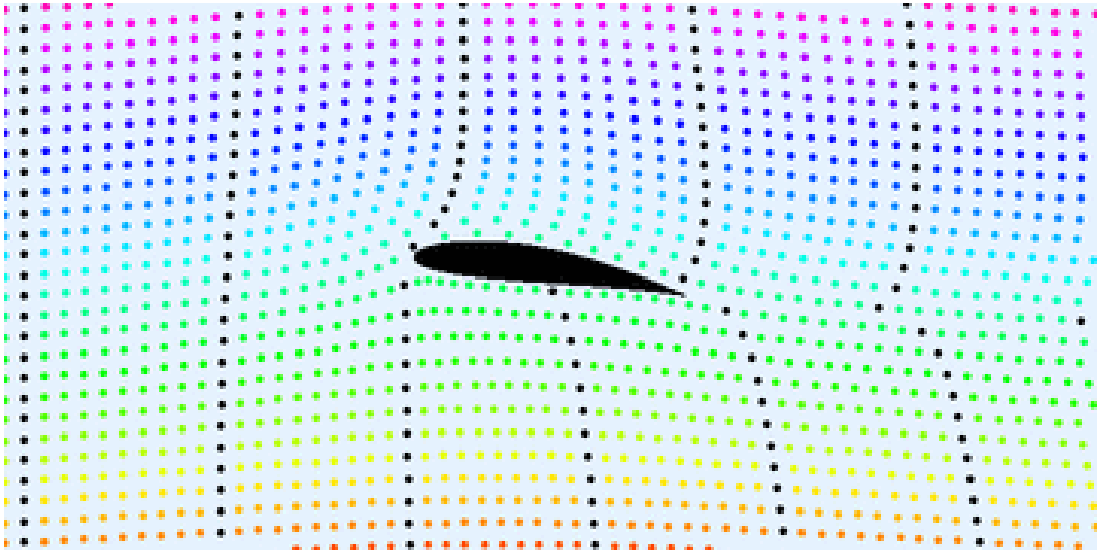
Magnus Effect



Magnus Effect

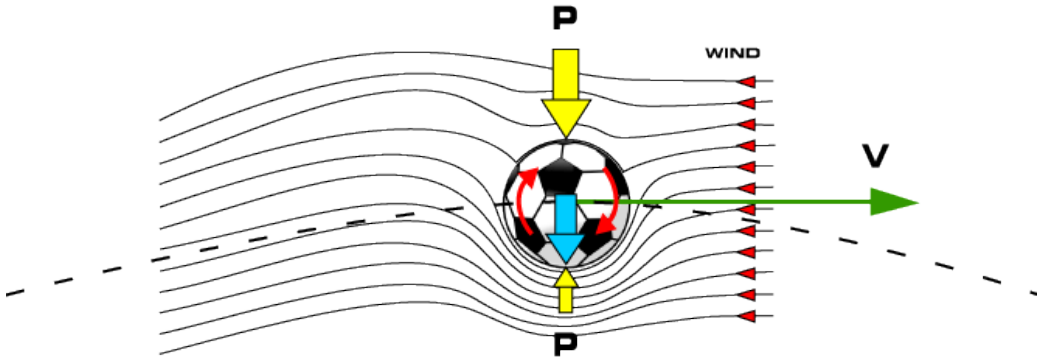
The overall behavior is similar to that around an aerofoil with a circulation which is generated by the mechanical rotation, rather than by airfoil action.

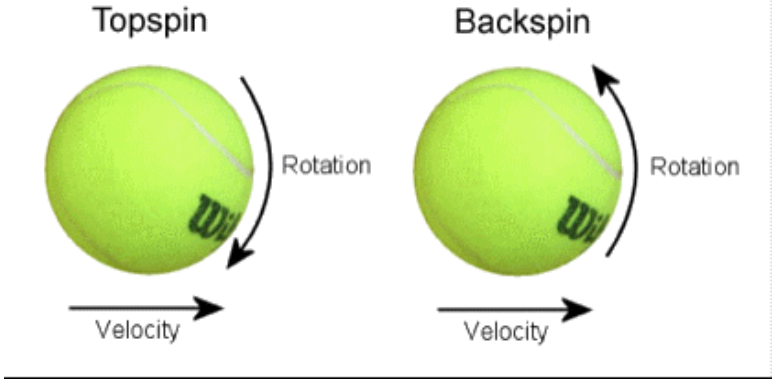
The airfoil is a Kármán–Trefftz airfoil, with parameters $\mu_x = -0.08$, $\mu_y = +0.08$ and $n = 1.94$. The angle of attack is 8°, and the flow is a potential flow



Magnus Effect

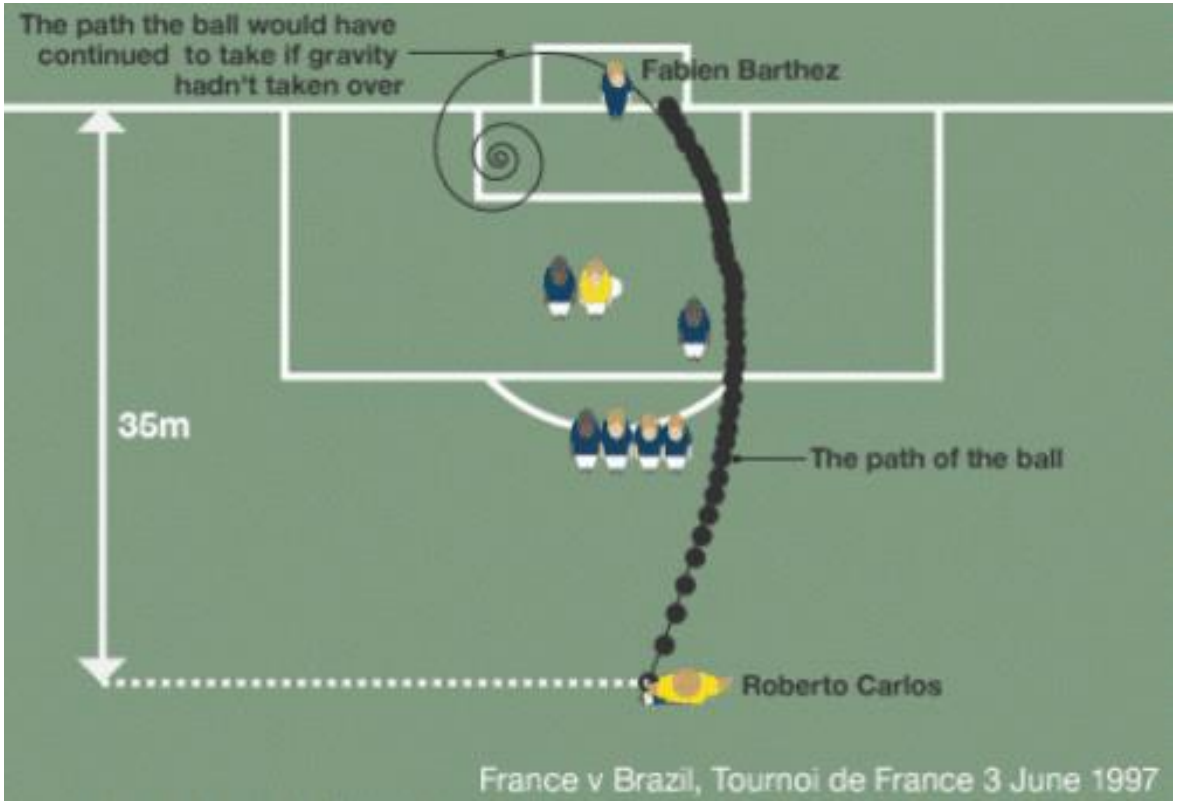
In terms of ball games, **topspin** is defined as spin about an axis perpendicular to the direction of travel, where the top surface of the ball is moving forward with the spin. Under the Magnus effect, topspin produces a downward swerve of a moving ball, greater than would be produced by gravity alone, and backspin has the opposite effect.



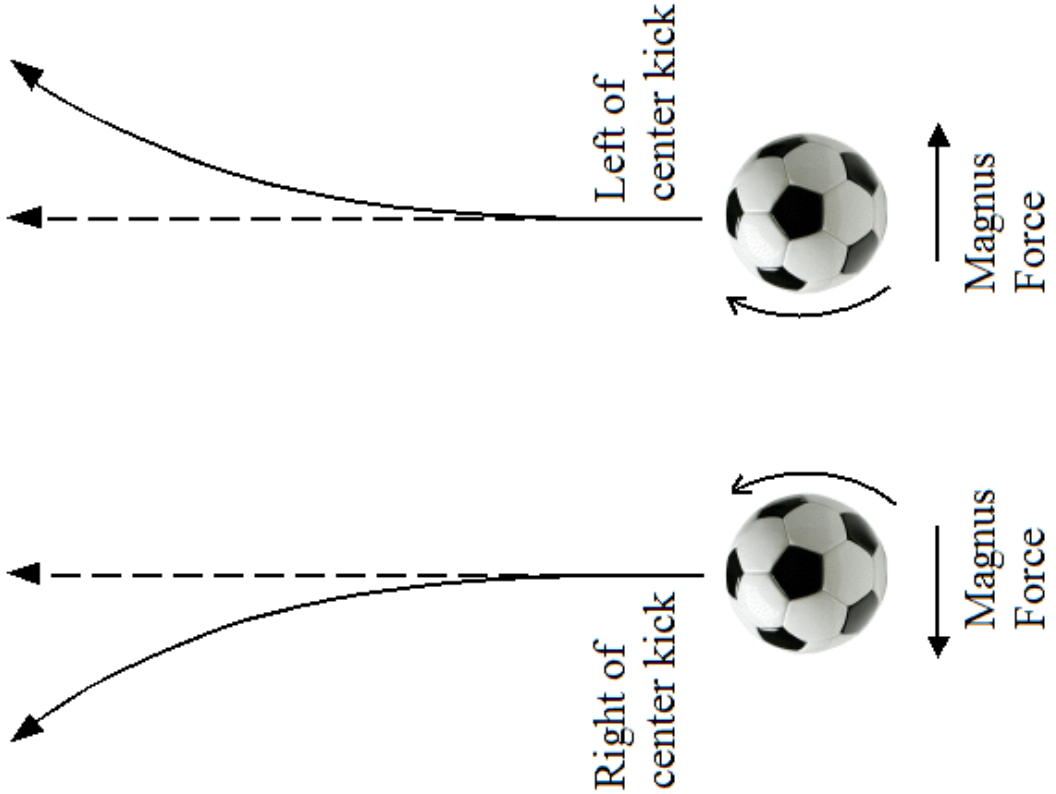


The Best Example of the Magnus Effect

The Best Example of the Magnus Effect



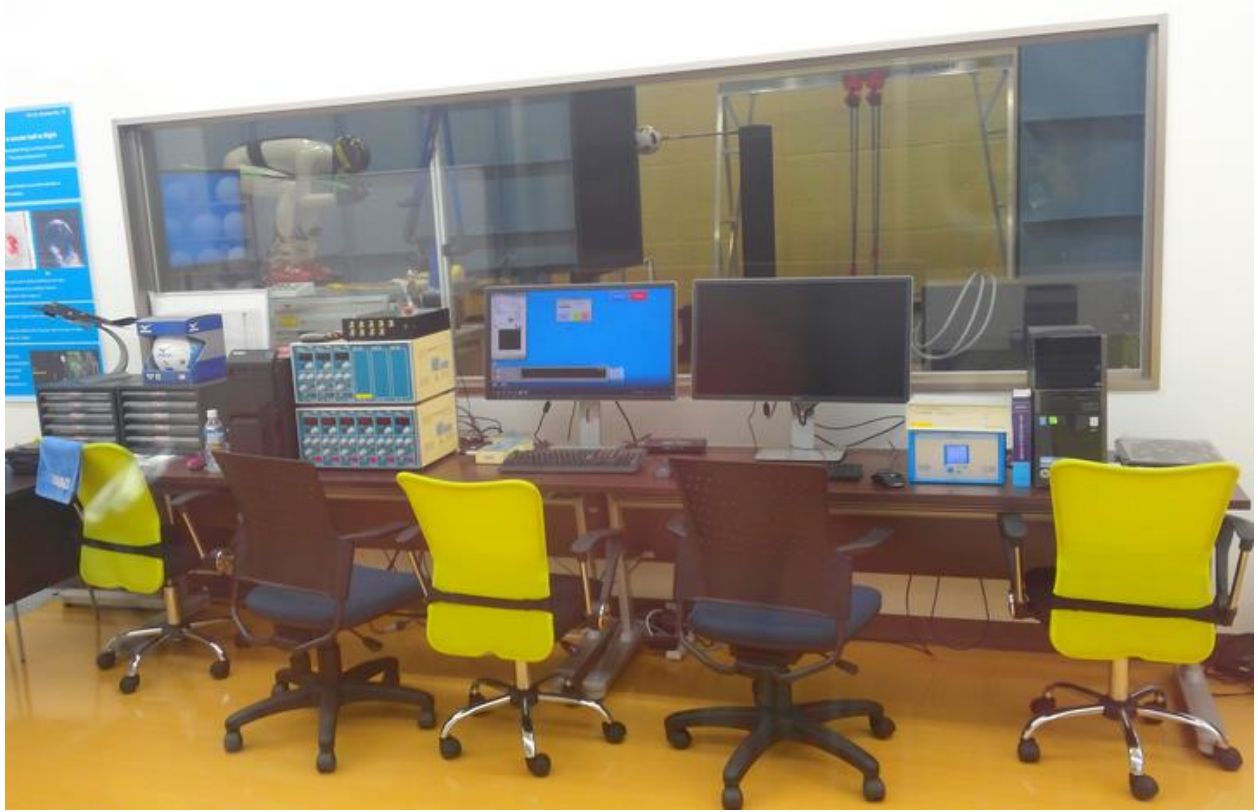
Magnus Effect



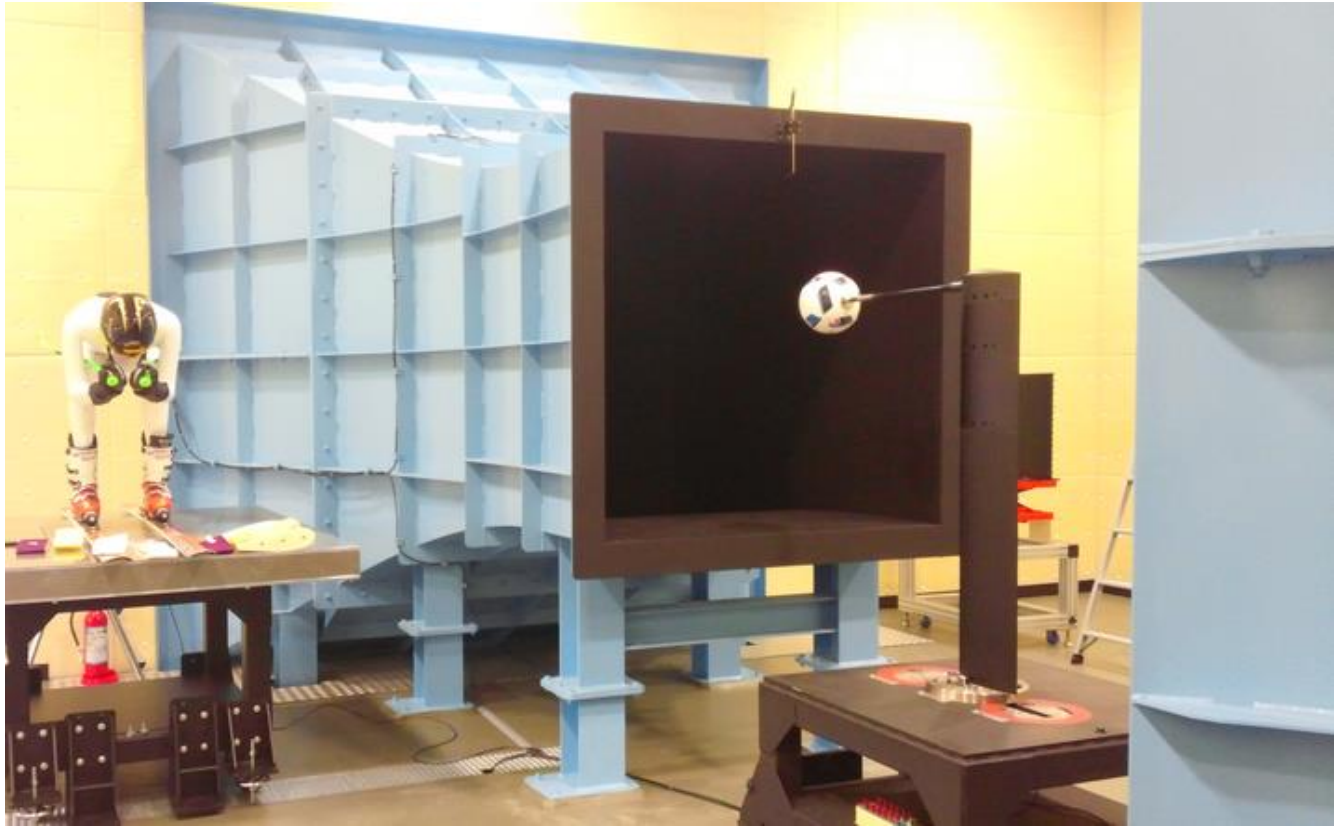
Soccer Balls

筑波大学 University of Tsukuba

Soccer Balls

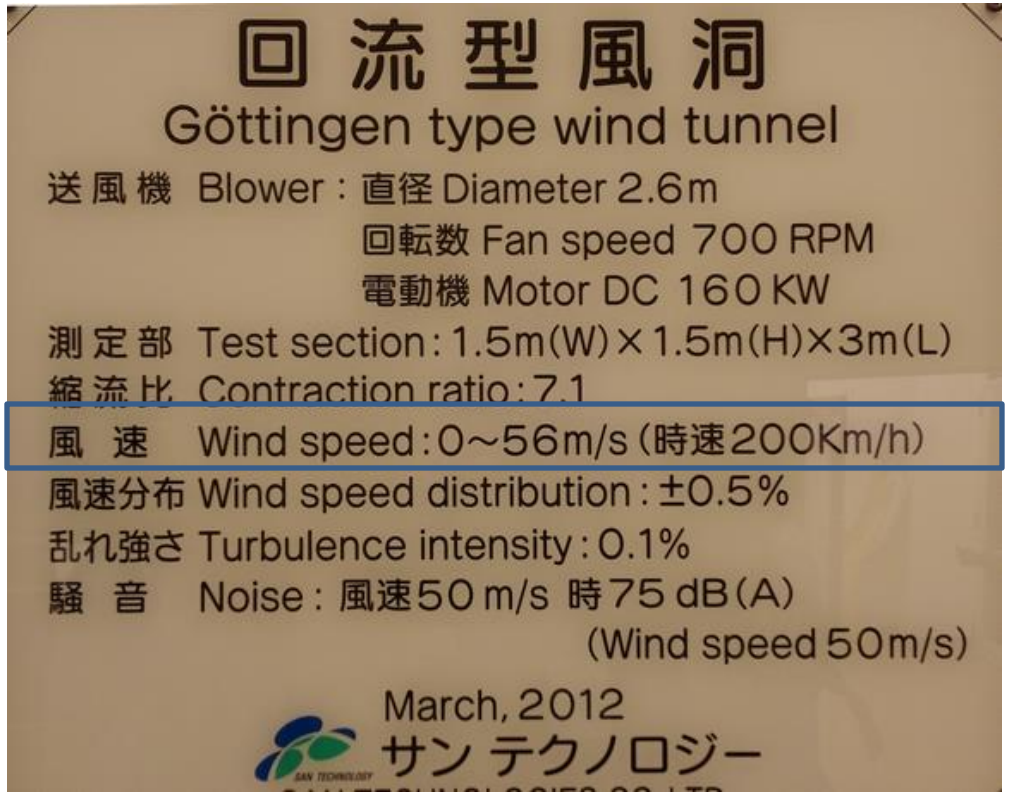


Soccer Balls

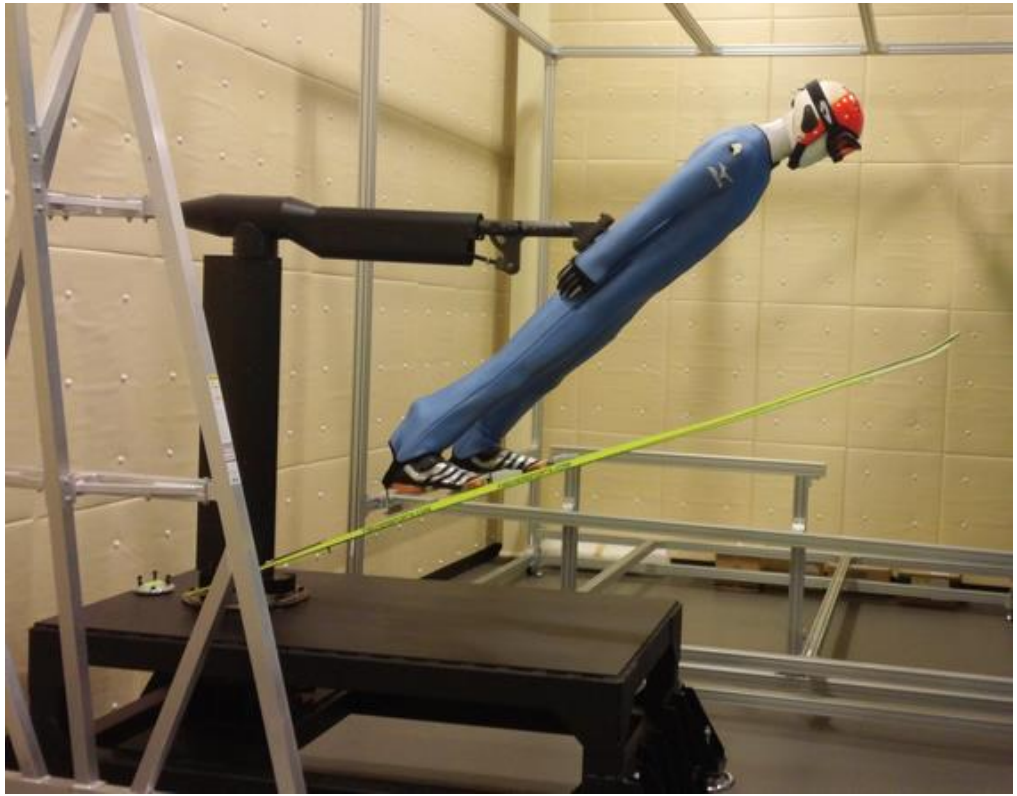


筑波大学 University of Tsukuba

Soccer Balls



Soccer Balls

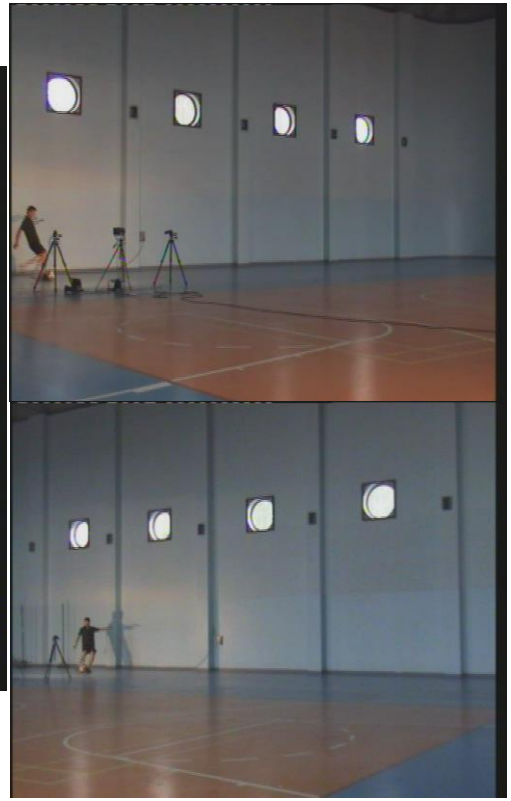


筑波大学 University of Tsukuba

Soccer Balls

Soccer Balls

- trajectory of the ball



The trajectory of the soccer ball were measured by using three video cameras that were operating at PAL standart (50Hz- interlaced).

Soccer Balls

- spin of the ball
- Initial Velocity
- Angle of Release

The initial conditions of the soccer ball were measured by using two high-speed video cameras that were operating at 500Hz.

Soccer Balls

- Cardan Angles

  x’y”z” (*)

  z’y”x”

  ...

- Euler Angles

  x’y”x”

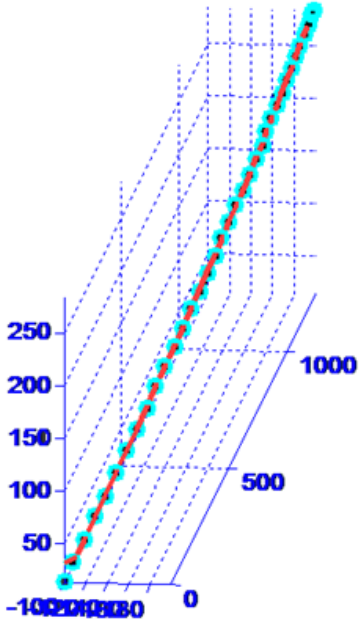
  z’y”z”

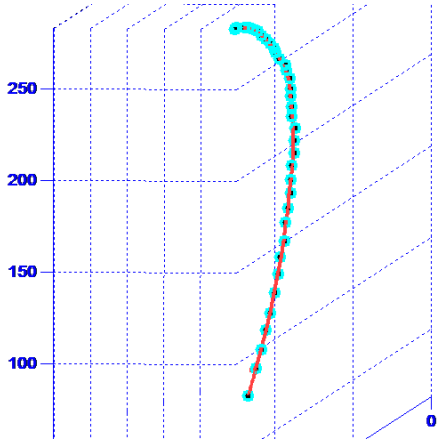
  ...

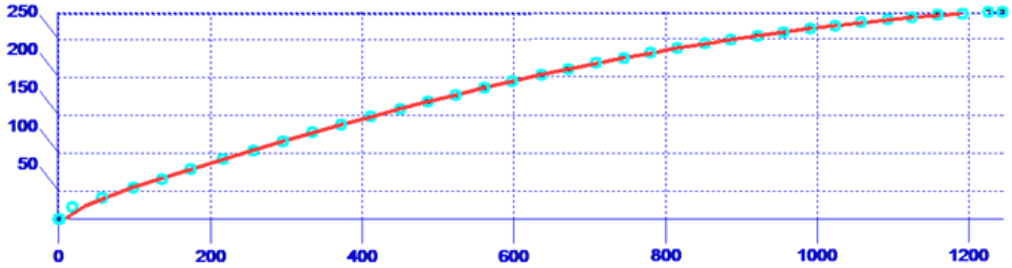
PHYSICS in COMPUTER ANIMATIONS and GAMES

- Experiment
- Simulation

Elevation Angle	10°
Azimuth Angle	12°
Drag Coefficient	0.27
Lift Coefficient	0.24
Spin Rate	1.1 Hz
Velocity	21 ms^{-1}







The Best Example of the Magnus Effect

Drag Coefficient	0.25
Lift Coefficient	0.23
Elevation Angle	14°
Azimuth Angle	38°
Initial Velocity of the Ball	36 ms^{-1}
Spin Rate	2.2 Hz

- Initial Values for Simulation

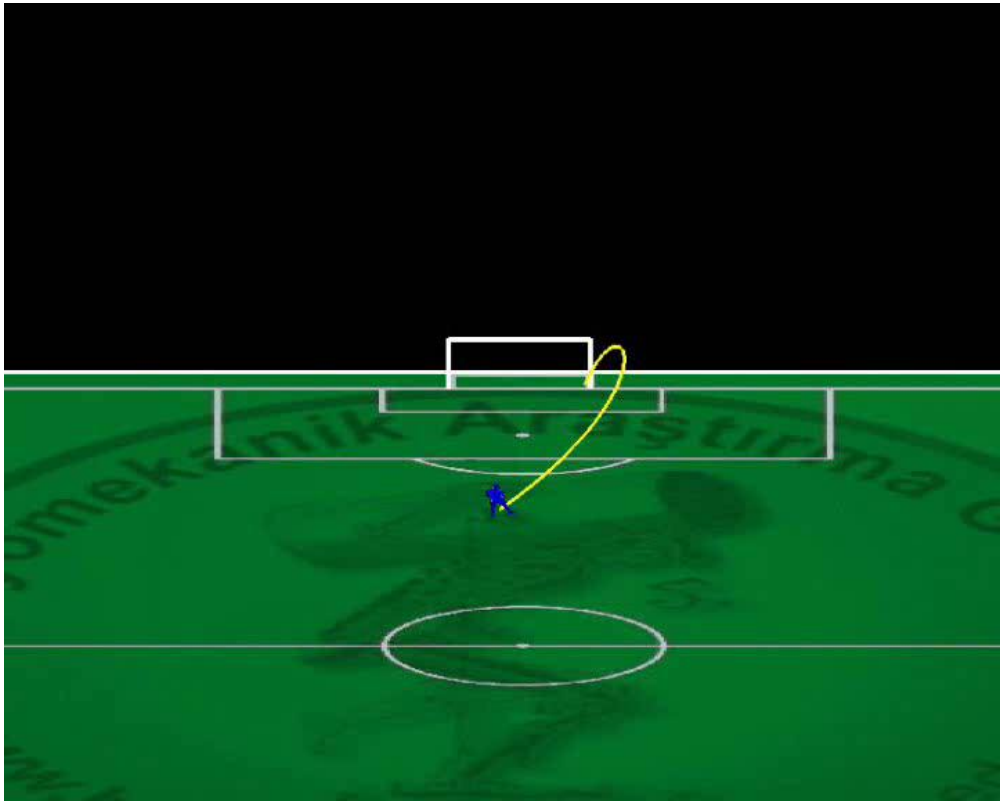
Local Gravity (Due to Latitude and Altitude)

Lyon	9.8061 $m.s^{-2}$	45° 44’ N	248m
İstanbul (Dolmabahçe)	9.8025 $m.s^{-2}$	40° 58’ N	10m
Ankara (Beytepe)	9.7984 $m.s^{-2}$	39° 57’ N	1030m

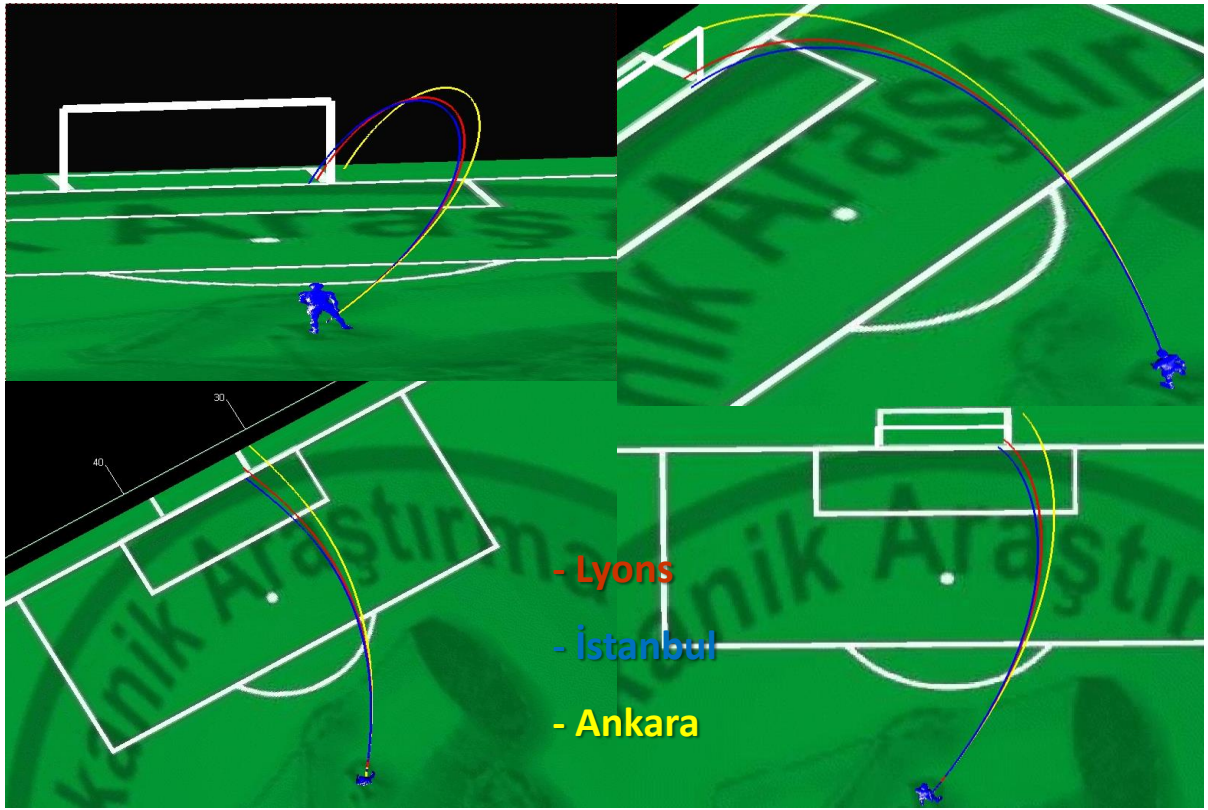
Local Air Density (Due to Temperature and Altitude)

Lyon	1.1670	20°C	248m
İstanbul (Dolmabahçe)	1.2031	20°C	10m
Ankara (Beytepe)	1.0559	20°C	1030m

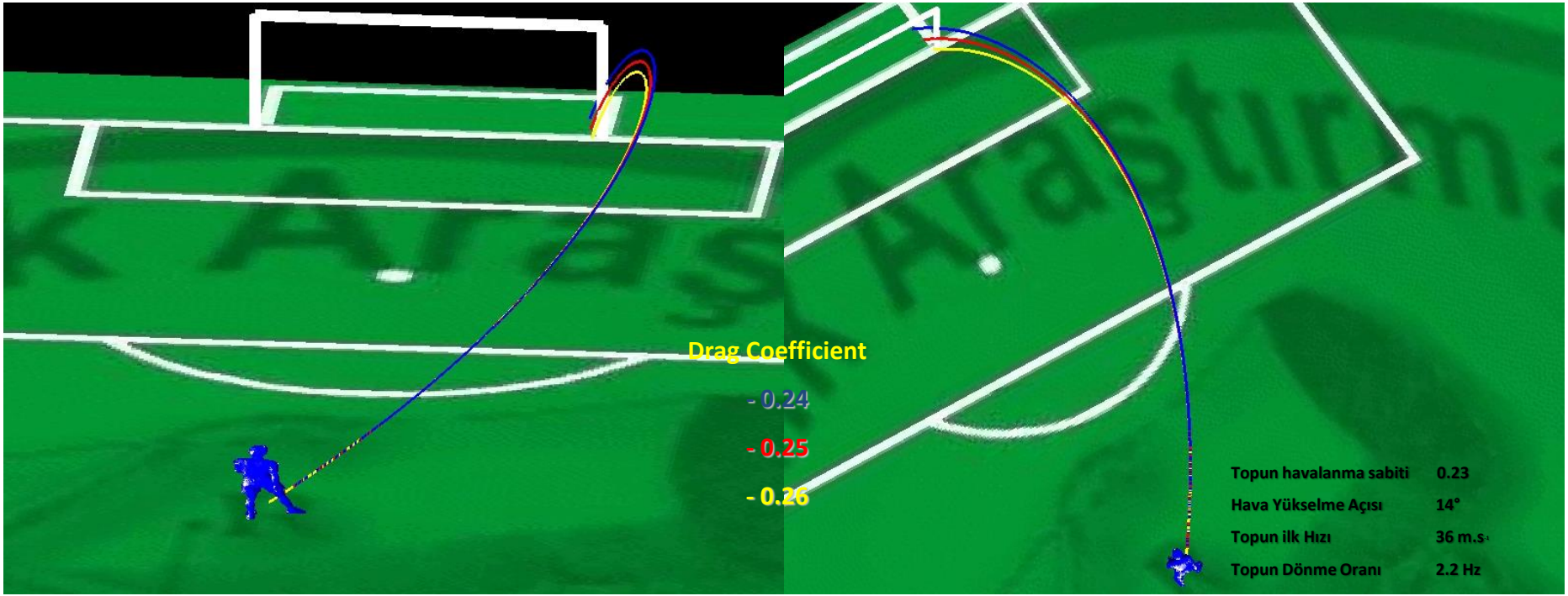
The Best Example of the Magnus Effect



The Best Example of the Magnus Effect

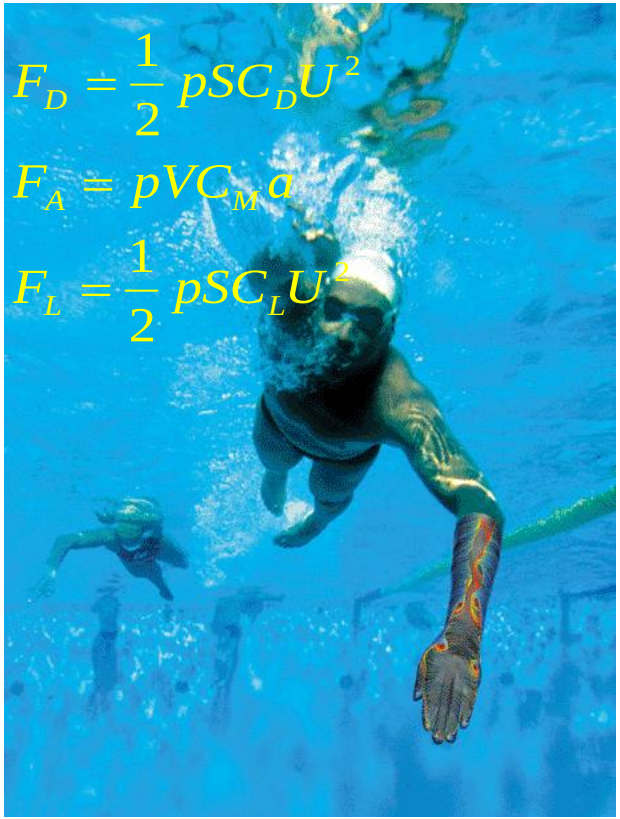


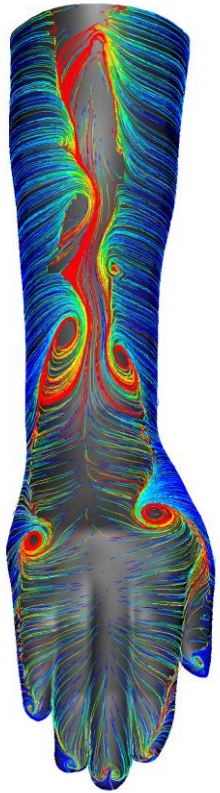
The Best Example of the Magnus Effect

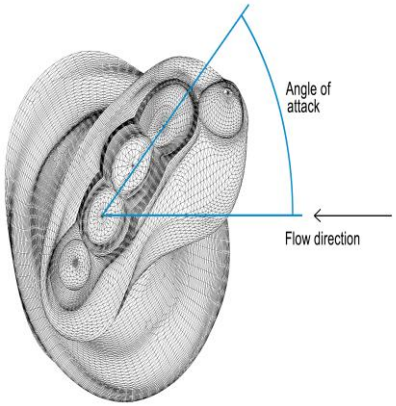


The Best Example of the Magnus Effect

PHYSICS in COMPUTER ANIMATIONS and GAMES

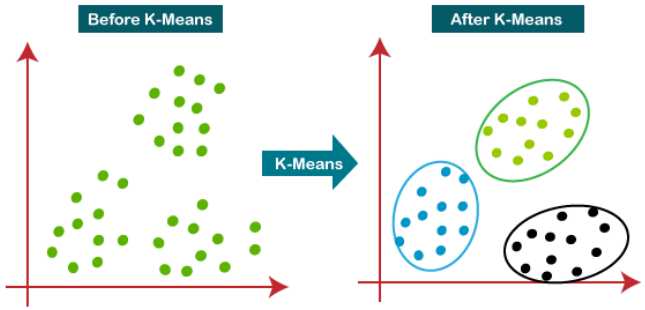


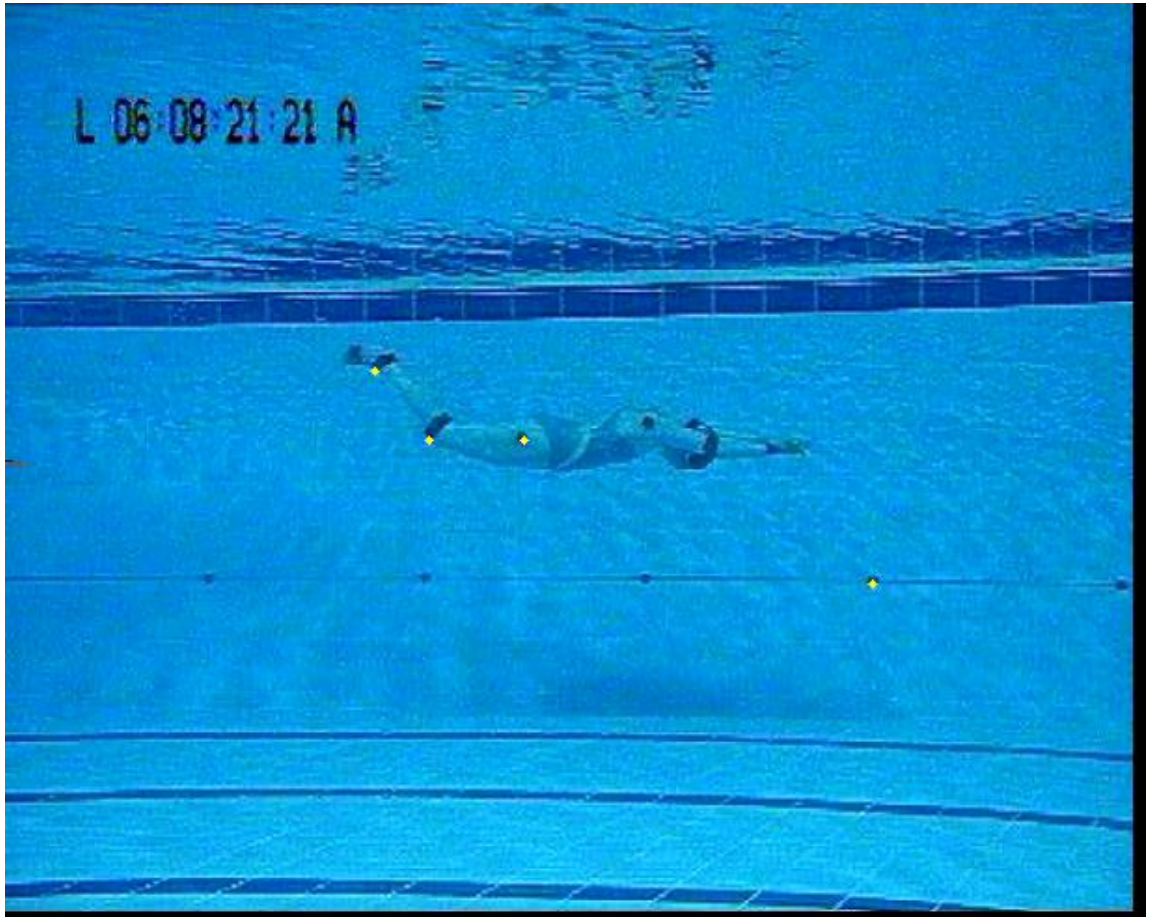




PHYSICS in COMPUTER ANIMATIONS and GAMES

Perfect Glide Experiment



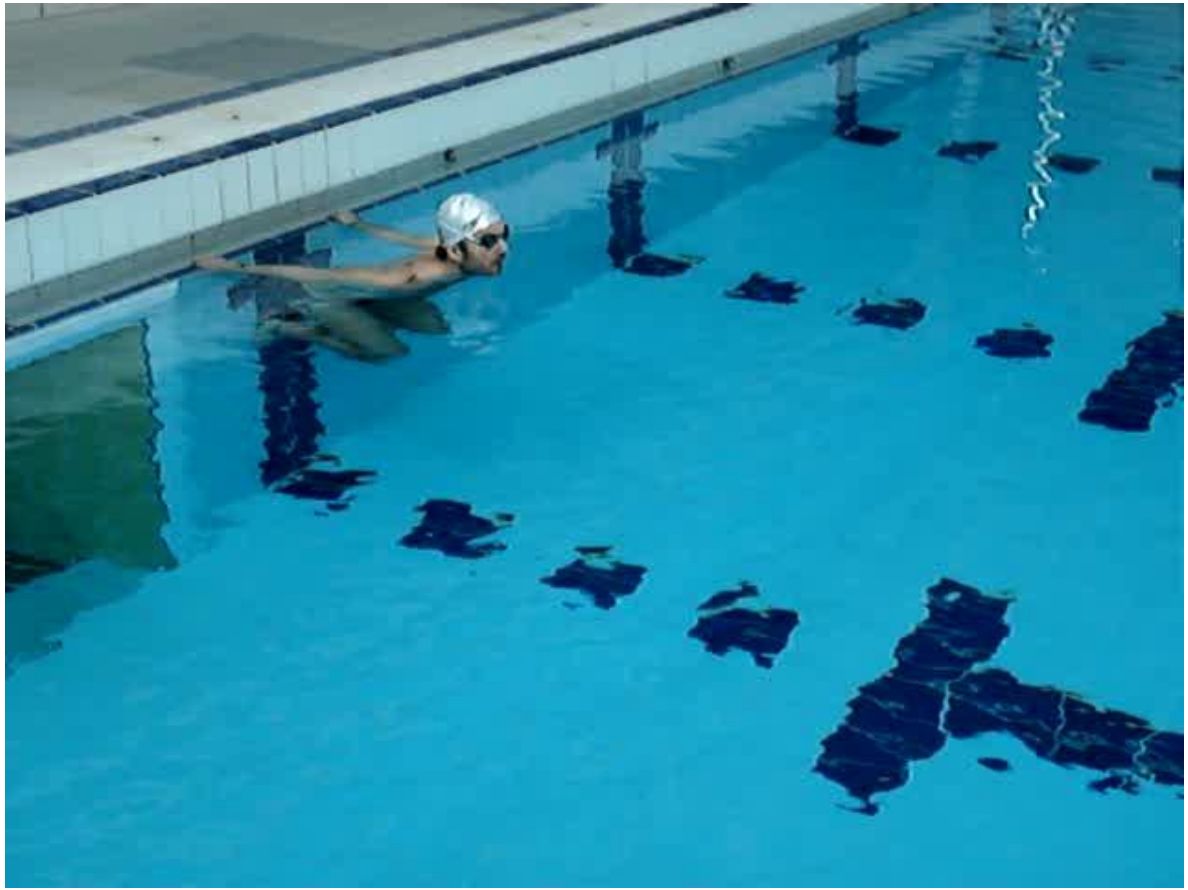


PHYSICS in COMPUTER ANIMATIONS and GAMES

Perfect Glide

PHYSICS in COMPUTER ANIMATIONS and GAMES

Perfect Glide Experiment

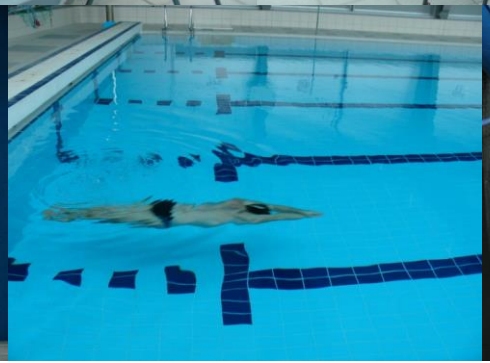


PHYSICS in COMPUTER ANIMATIONS and GAMES

Perfect Glide Experiment

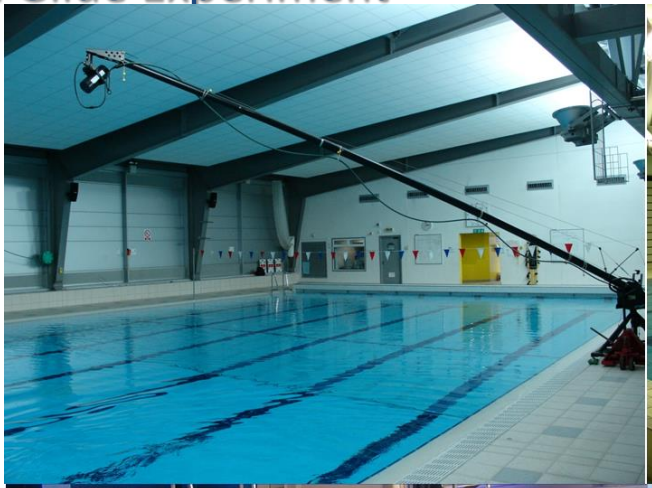
PHYSICS in COMPUTER ANIMATIONS and GAMES

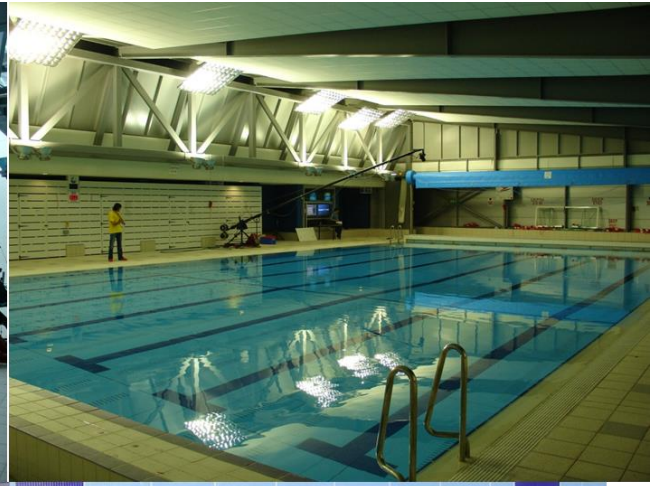
Perfect Glide Experiment



PHYSICS in COMPUTER ANIMATIONS and GAMES

Perfect Glide Experiment





Development of Immediate Feedback Software for Optimising Glide Performance and Time of Initiating Post-Glide Actions

Roozbeh Naemi[1], Serdar Aritan[2], Simon Goodwill[3], Steve Haake[3], Ross Sanders[1].

(1) : Centre for Aquatics Research and Education, The University of Edinburgh, St Leonard's Land, Holyrood Road, Edinburgh, UK EH8 8AQ
0044 131 651 4117 / 0044 131 651 6521
E-mail : Roozbeh.naemi@education.ed.ac.uk

(2) : Biomechanics Research Group, School of Sports Sciences and Technology, Beytepe, 06800,Ankara, Turkey
0090 312 297 6893 / 0090 312 299 2167
E-mail : serdar.aritan@hacettepe.edu.tr

(3) : Sports Engineering @ Centre for Sport and Exercise Science Sheffield Hallam University, Collegiate Hall Sheffield, UK S10 2BP
0044 114 225 4435
E-mail : s.r.goodwill@shu.ac.uk

(4) : Sports Engineering @ Centre for Sport and Exercise Sciences Sheffield Hallam University, Collegiate Crescent, Sheffield, UK S10 2BP
0044 114 225 2429 / *0044* 114 225 4356
E-mail : S.J.Haake@shu.ac.uk

(5) : Centre for Aquatics Research and Education, The University of Edinburgh, St Leonard's Land Holyrood Road, Edinburgh, UK EH8 8AQ
0044 131 651 6580 / 0044 131 6516521
E-mail : r.sanders@ed.ac.uk

TOPICS: Performance Sports, Biomechanics, Measurement Systems

Abstract: Performance in starts and turns is a major contributor to success in swimming and is influenced greatly by the glide efficiency and the timing of commencing the post-glide action (including kick in all strokes and the underwater pull in breaststroke starts and turns). The main aim of this research is to develop and test 'user friendly' software for providing immediate feedback to swimmers and coaches to optimise glide performance and time of initiating post-glide actions in starts, turns, and the glide phase of the breaststroke.

Medal table

Last updated after the 2020 Summer Olympics

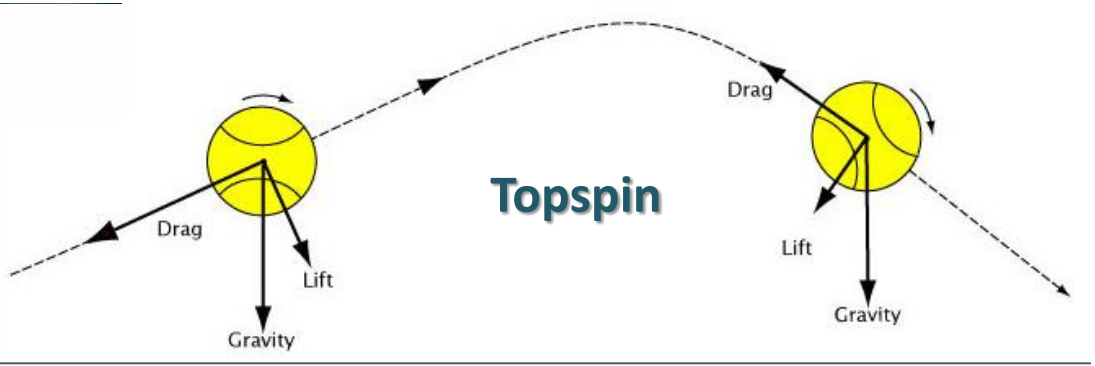
Rank	Nation	Gold	Silver	Bronze	Total
1	United States (USA)	257	178	144	579
2	Australia (AUS)	69	70	73	212
3	*East Germany (GDR)*	38	32	22	92
4	Hungary (HUN)	28	26	20	74
5	Japan (JPN)	24	27	32	83
6	Great Britain (GBR)	20	29	30	79
7	Netherlands (NED)	19	20	19	58
8	China (CHN)	16	21	12	49
9	Germany (GER)	13	18	30	61
10	*Soviet Union (URS)*	12	21	26	59

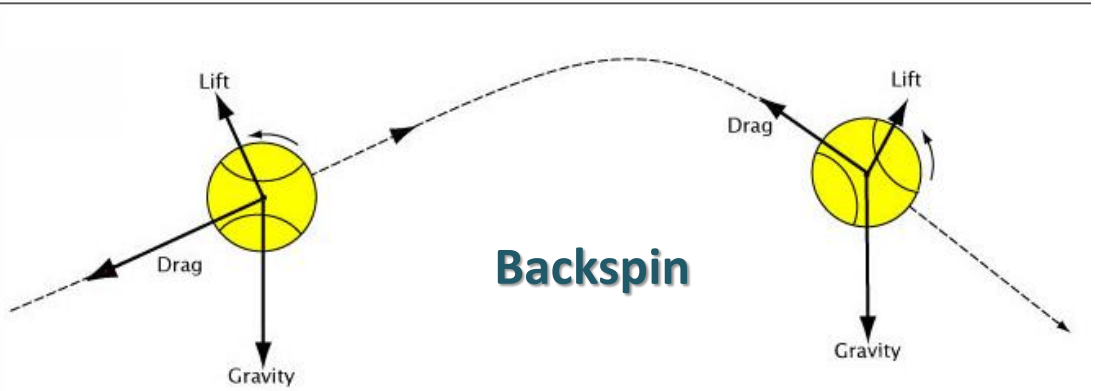
Medal table

Retrieved from 2008 NBC Olympics website.

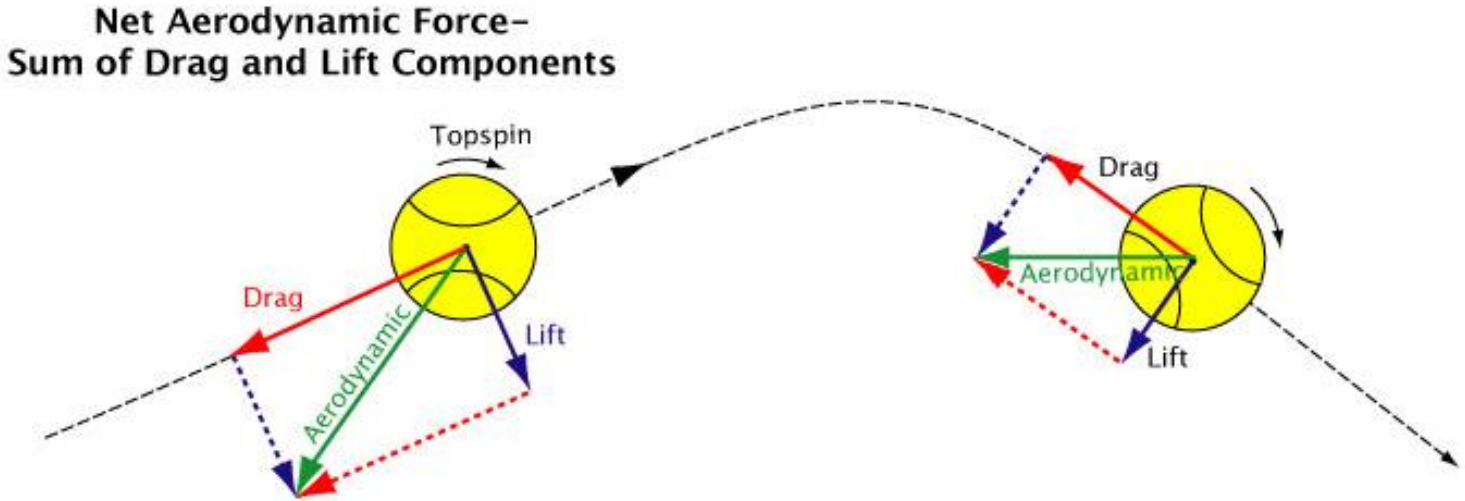
Rank	Nation	Gold	Silver	Bronze	Total
1	United States (USA)	12	9	10	31
2	Australia (AUS)	6	6	8	20
3	Great Britain (GBR)	2	2	2	6
4	Japan (JPN)	2	0	3	5
5	Germany (GER)	2	0	1	3
6	Netherlands (NED)	2	0	0	2
7	China (CHN)	1	3	2	6
8	Zimbabwe (ZIM)	1	3	0	4
9	France (FRA)	1	2	3	6
10	Russia (RUS)	1	1	2	4

Magnus Effect

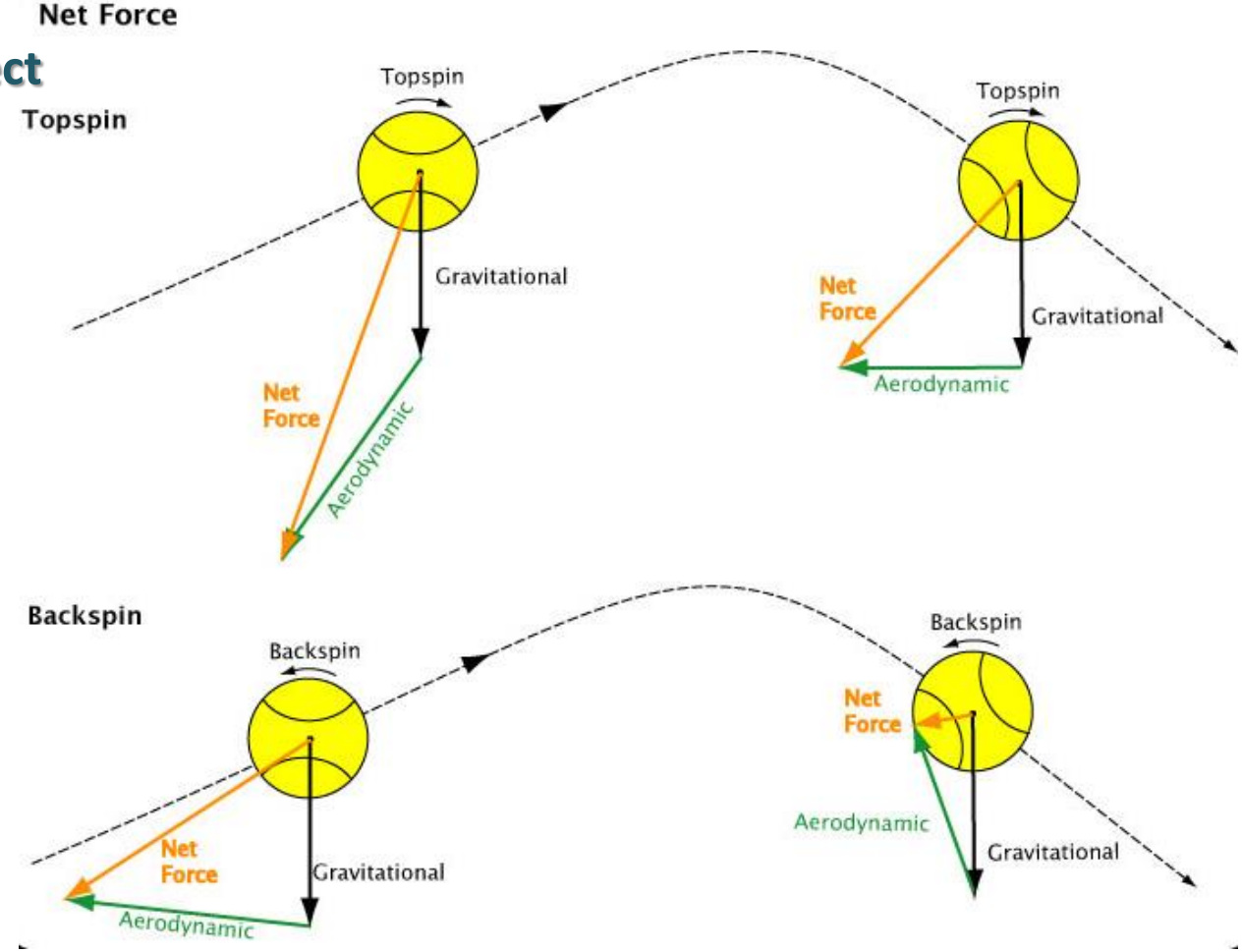




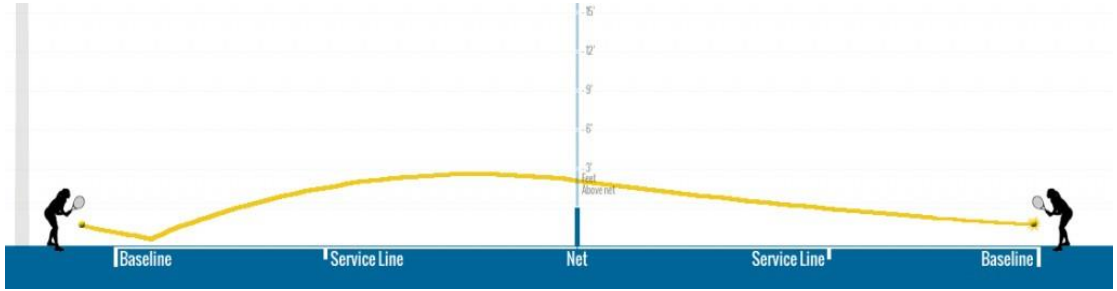
Magnus Effect

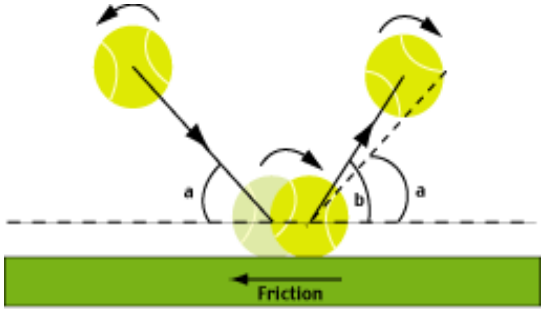


Magnus Effect

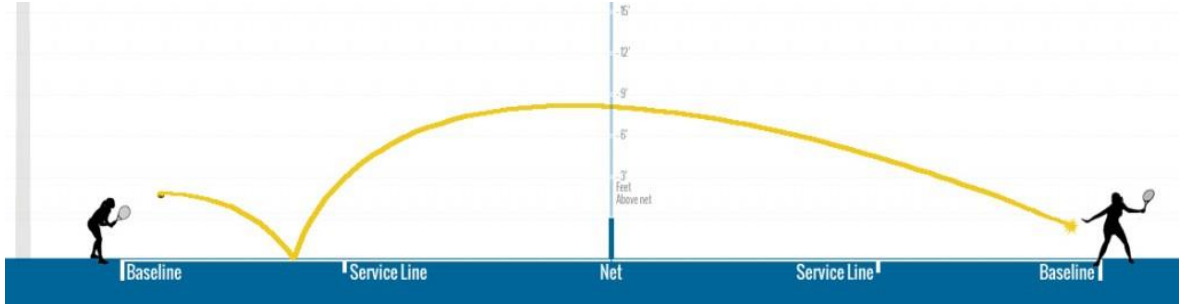


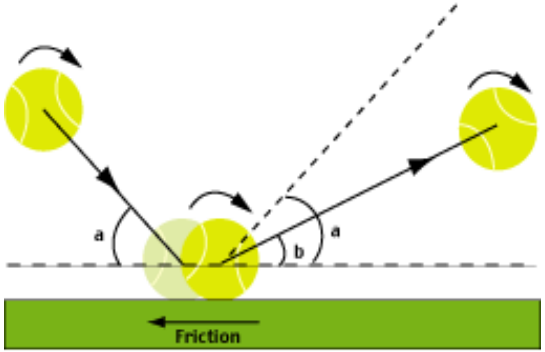
PHYSICS in COMPUTER ANIMATIONS and GAMES



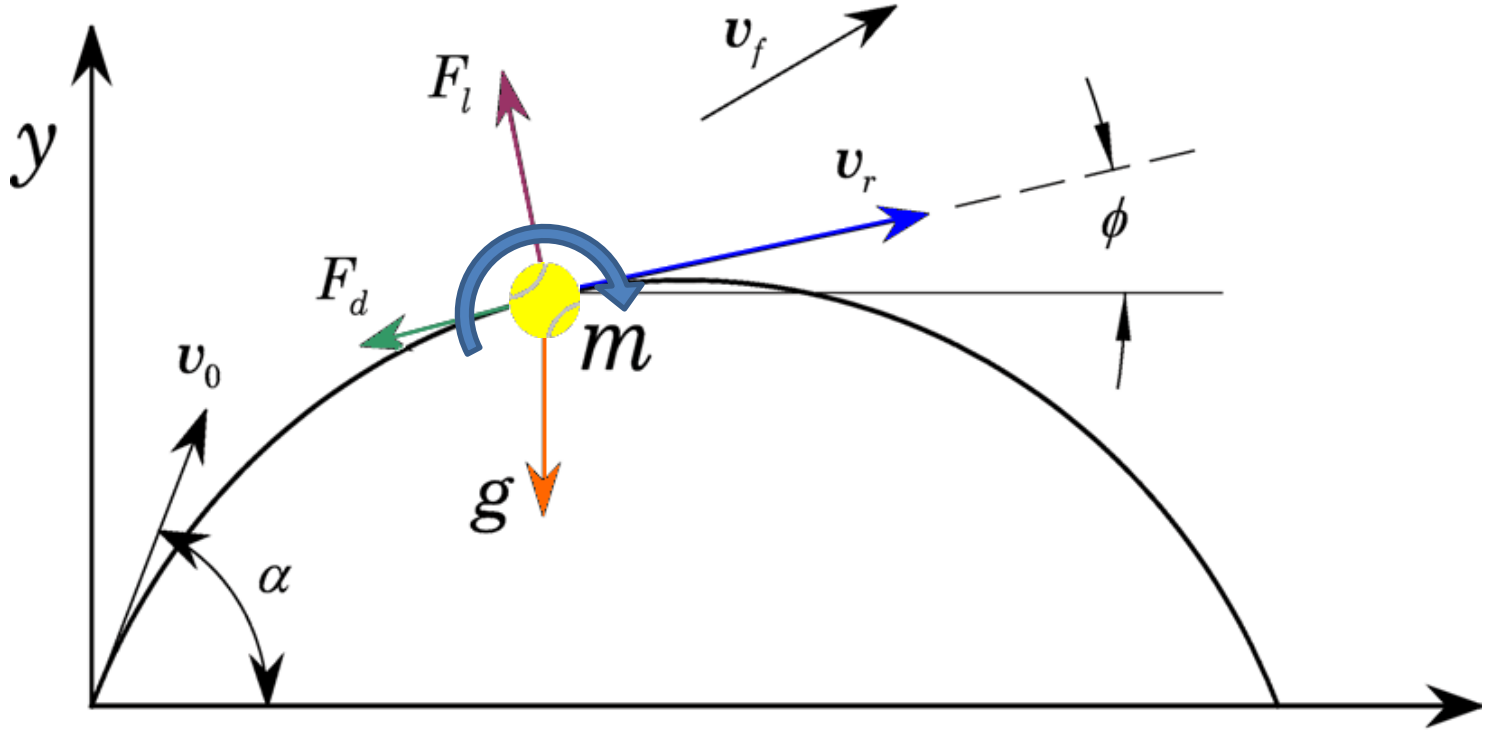


Incident with Back Spin





Incident with Top Spin



F_l is the lift force stemming from the rotation of the ball (the Magnus-effect) and is normal to v_r. With the given direction the ball rotates counter-clockwise (backspin). F_d is the fluids resistance against the motion and is parallel to v_r.

These forces are given by

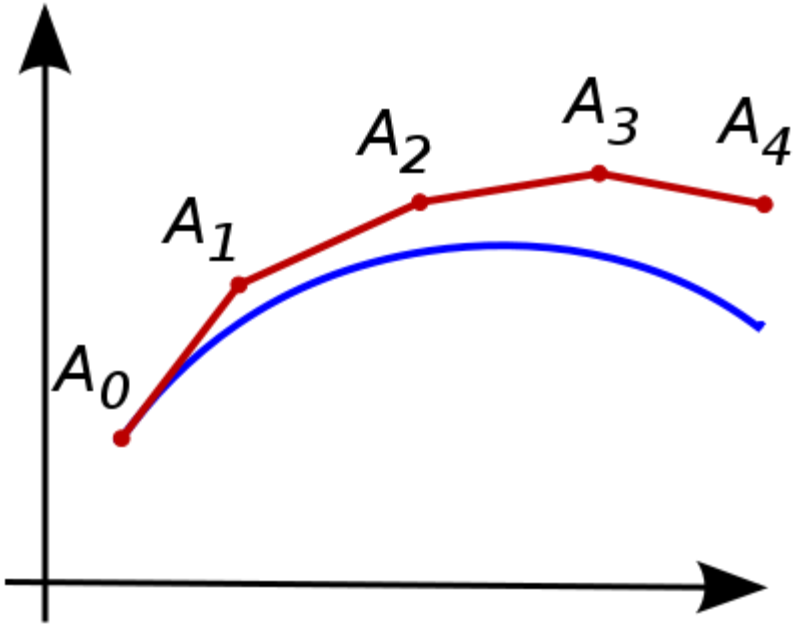
$$F_d = \frac{1}{2}\rho_f A C_D v_r^2$$

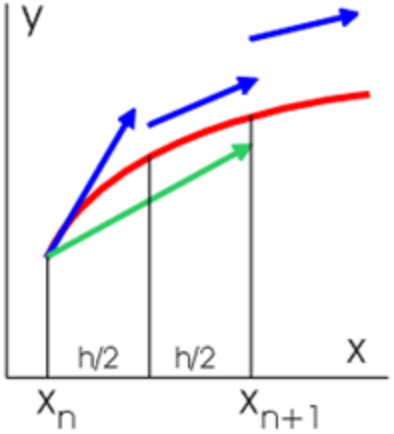
$$F_l = \frac{1}{2}\rho_f A C_L v_r^2$$

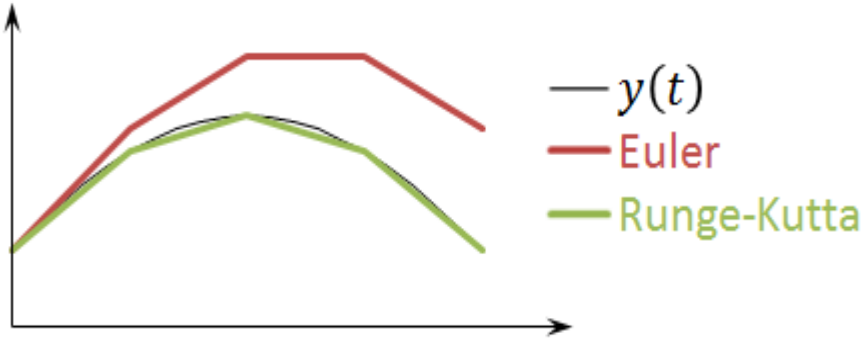
$\mathbf{C_D}$ is the **drag** coefficient, $\mathbf{C_L}$ is the **lift** coefficient, **A** is the area projected in the velocity direction and $\boldsymbol{\rho_f}$ is the density of the fluid. Newton's second law in x- and y-directions gives

$$\frac{dv_x}{dt} = -\rho_f \frac{A}{2m} v_r^2 (C_D \cos\varphi + C_L \sin\varphi)$$

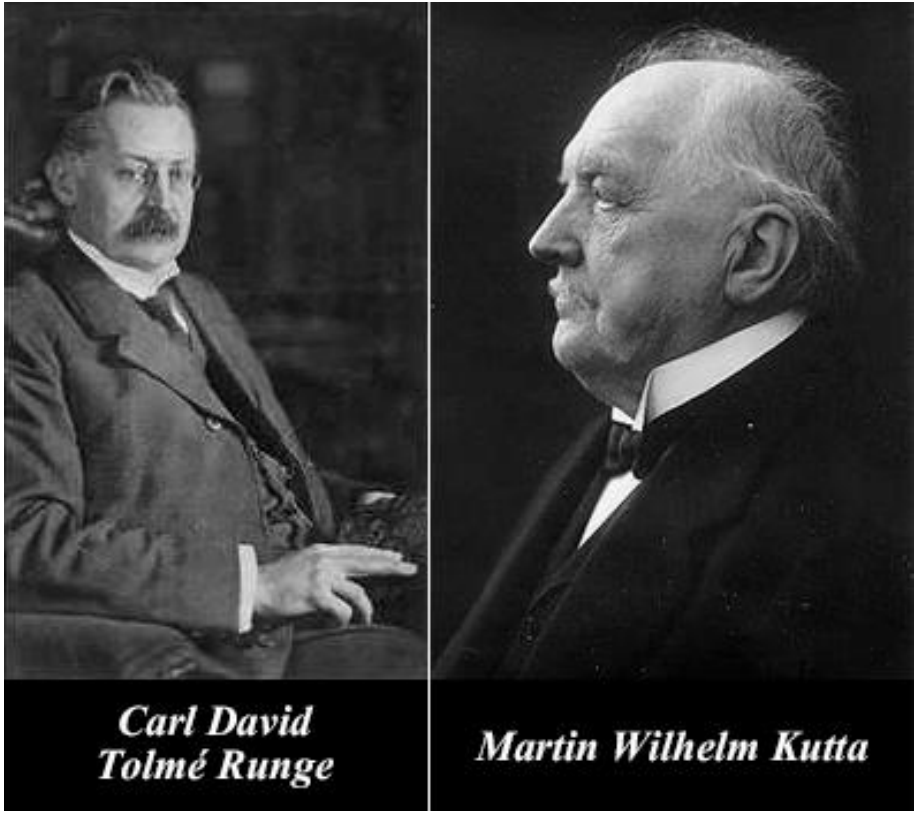
$$\frac{dv_y}{dt} = \rho_f \frac{A}{2m} v_r^2 (C_L \cos\varphi - C_D \sin\varphi) - g$$


A_0
A_1
A_2
A_3
A_4


y
h/2
h/2
x
x_n
x_{n+1}

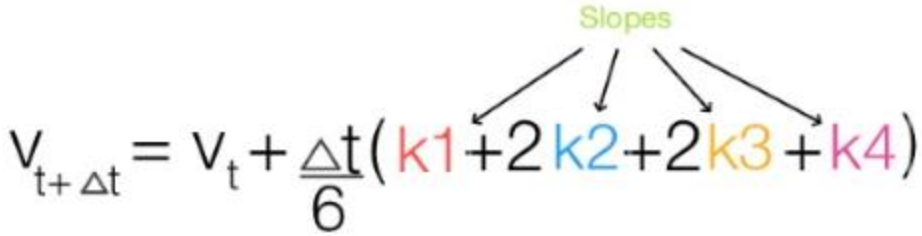

$y(t)$
Euler
Runge-Kutta

Carl David Tolme Runge – Martin Wilhelm Kutta



Runge –Kutta

The Runga-Kutta uses these slopes as weighted average to better approximate the actual slope, velocity, of the object.



PHYSICS in COMPUTER ANIMATIONS and GAMES

```
import numpy as np
import matplotlib.pyplot as plt

g = 9.81        # Gravity [m/s^2]
nu = 1.5e-5     # Kinematical viscosity [m^2/s]
rho_f = 1.29    # Density of fluid [kg/m^3]
rho_s = 418     # Density of sphere [kg/m^3]
d = 67.0e-3     # Diameter of the sphere [m]
v0 = 50.0       # Initial velocity [m/s]
vfx = 0.0       # x-component of fluid's velocity
vfy = 0.0       # y-component of fluid's velocity
CD = 0.55       # CD is typically about 0.5 - 0.6
CL = 0.3        # CL is normally taken to be positive for
                # backspin and negative for topspin.
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
def rk4(func, z0, time):
    """The Runge-Kutta 4 scheme for solution of systems of ODEs.
    z0 is a vector for the initial conditions,
    the right hand side of the system is represented by func which returns
    a vector with the same size as z0 ."""

    z = np.zeros((size(time), size(z0)))
    z[0, :] = z0

    for i, t in enumerate(time[0:-1]):
        dt = time[i+1] - time[i]
        dt2 = dt/2.0
        k1 = np.asarray(func(z[i, :], t))                  # predictor step 1
        k2 = np.asarray(func(z[i, :] + k1*dt2, t + dt2))   # predictor step 2
        k3 = np.asarray(func(z[i, :] + k2*dt2, t + dt2))   # predictor step 3
        k4 = np.asarray(func(z[i, :] + k3*dt, t + dt))     # predictor step 4
        z[i+1, :] = z[i, :] + dt/6.0*(k1 + 2.0*k2 + 2.0*k3 + k4)# Corrector step
    return z
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

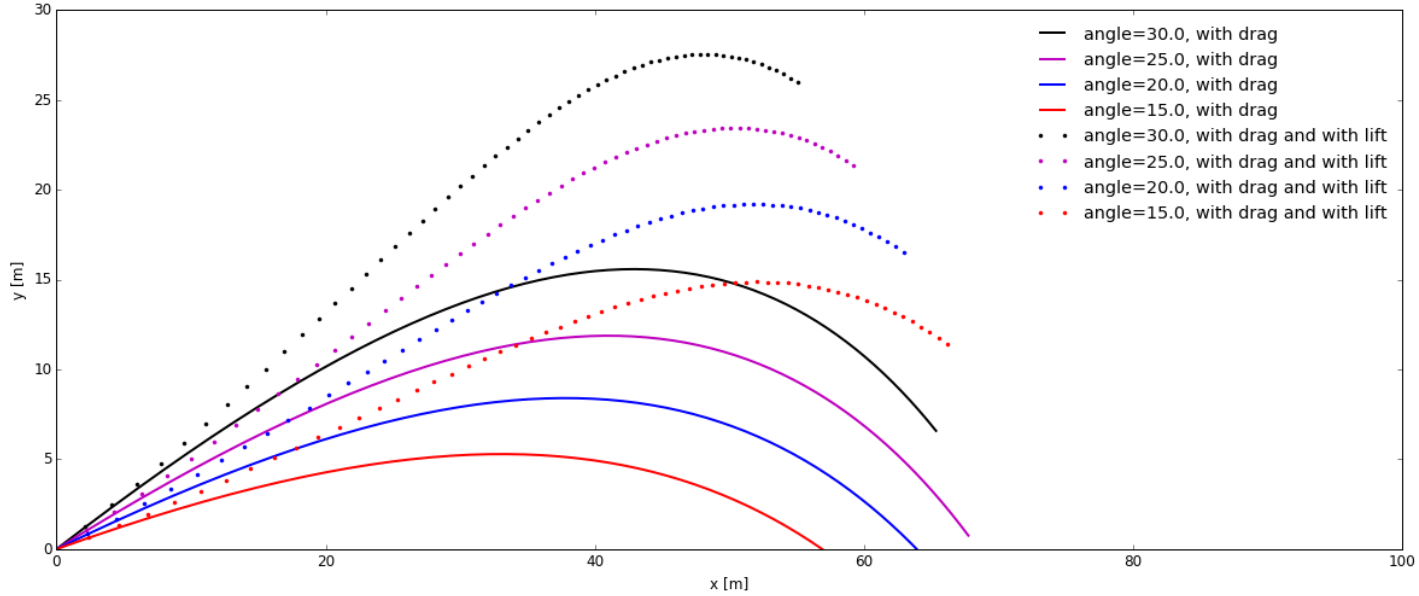
```
# tennis ball without lift
def f2(z, t):
    """4x4 system for golf ball with drag in two directions."""
    zout = np.zeros_like(z)
    C = 3.0*rho_f/(4.0*rho_s*d)
    vrx = z[2] - vfx
    vry = z[3] - vfy
    vr = np.sqrt(vrx**2 + vry**2)
    zout[:] = [z[2], z[3], -C*vr*(CD*vrx), C*vr*(-CD*vry) - g]
    return zout
# tennis ball with lift
def f3(z, t):
    """4x4 system for golf ball with drag and lift in two directions."""
    zout = np.zeros_like(z)
    C = 3.0*rho_f/(4.0*rho_s*d)
    vrx = z[2] - vfx
    vry = z[3] - vfy
    vr = np.np.sqrt(vrx**2 + vry**2)
    zout[:] = [z[2], z[3], -C*vr*(CD*vrx + CL*vry), C*vr*(CL*vrx - CD*vry) - g]
    return zout
```

```
# main program starts here

T = 3   # end of simulation
N = 60  # no of time steps
time = np.linspace(0, T, N+1)
N2 = 4
alfa = np.linspace(30, 15, N2)      # Angle of elevation [degrees]
angle = alfa*np.pi/180.0            # convert to radians

legends = []
line_color = ['k', 'm', 'b', 'r']
fig, ax = plt.subplots(figsize = (20, 8))# width, height in inches
```

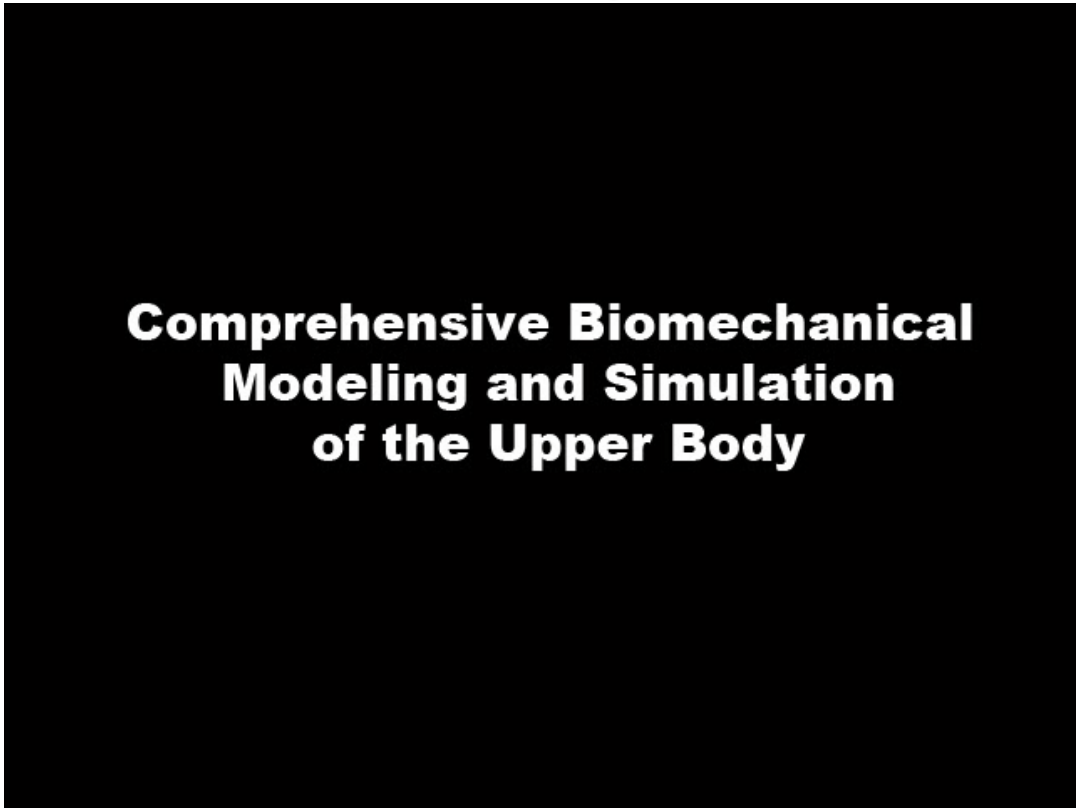
PHYSICS in COMPUTER ANIMATIONS and GAMES

```
# computing and plotting
# tennis ball with drag
for i in range(0, N2):
    z0 = np.zeros(4)
    z0[2] = v0*np.cos(angle[i])
    z0[3] = v0*np.sin(angle[i])
    z = rk4(f2, z0, time)
    ax.plot(z[:, 0], z[:, 1], '-', color=line_color[i])
    legends.append('angle=' + str(alfa[i]) + ', with drag')

# tennis ball with drag and lift
for i in range(0, N2):
    z0 = np.zeros(4)
    z0[2] = v0*np.cos(angle[i])
    z0[3] = v0*np.sin(angle[i])
    z = rk4(f3, z0, time)
    ax.plot(z[:, 0], z[:, 1], '.', color=line_color[i])
    legends.append('angle=' + str(alfa[i]) + ', with drag and with lift')
```

```
ax.legend(legends, loc='best', frameon=False)
ax.xlabel('x [m]')
ax.ylabel('y [m]')
ax.axis([0, 100, 0, 30])
Plt.show()
```

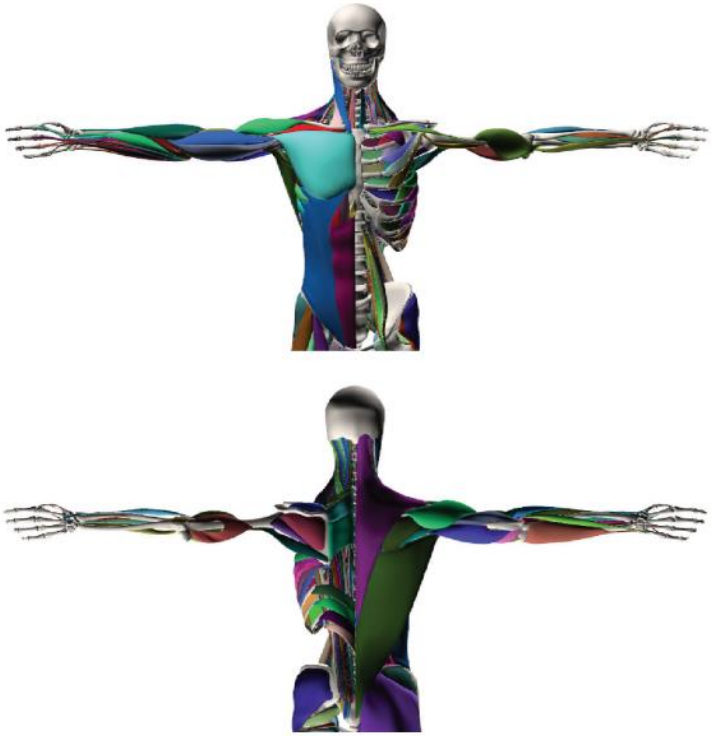


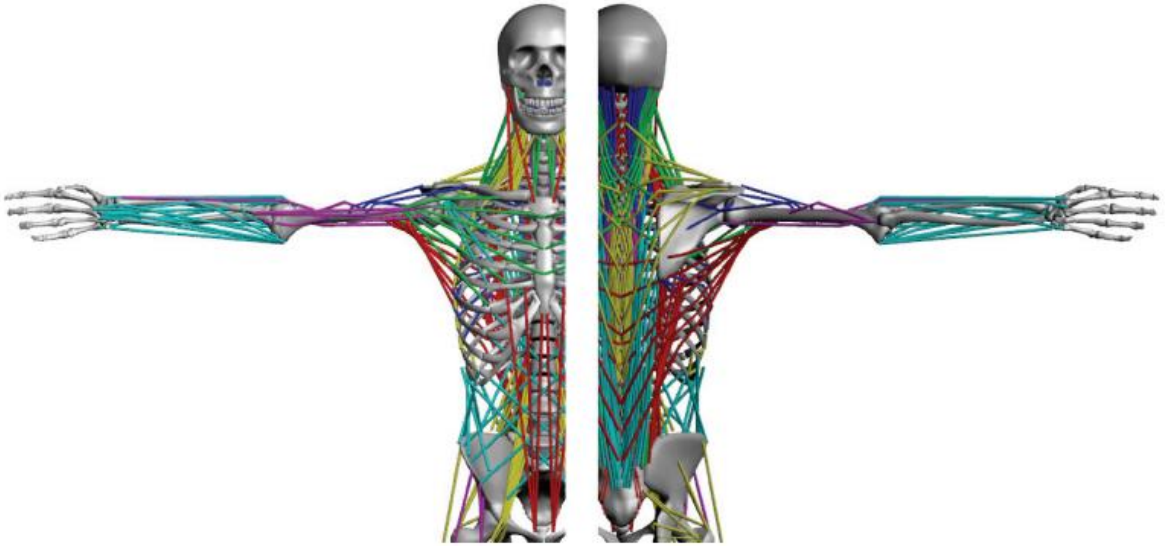
Comprehensive Biomechanical Modeling and Simulation of the Upper Body



Lee, S.-H., Sifakis, E., and Terzopoulos, D. 2009. Comprehensive biomechanical modeling and simulation of the upper body. ACM Trans. Graph. 28, 4, Article 99 (August 2009)

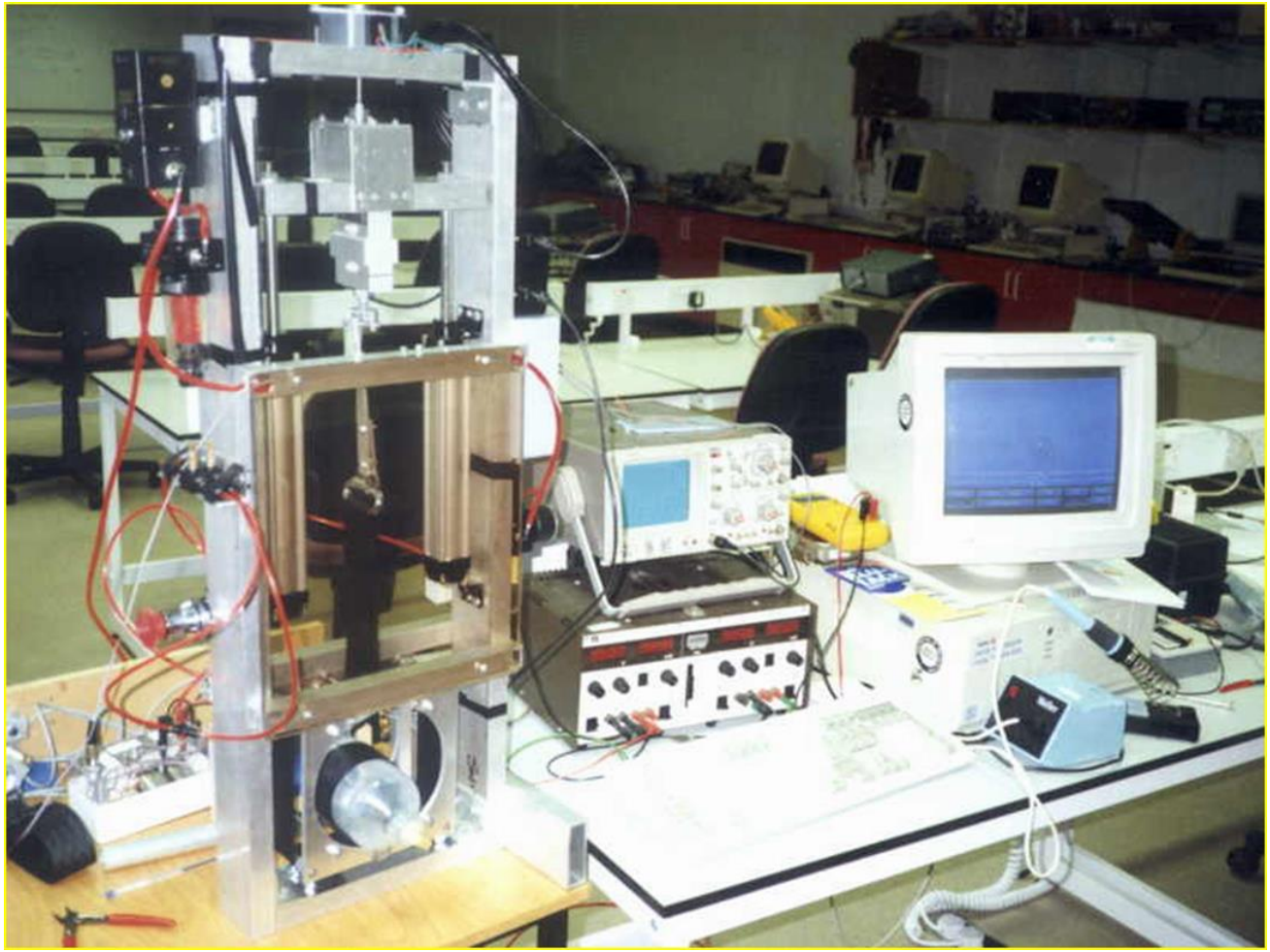
Physically Based Modeling

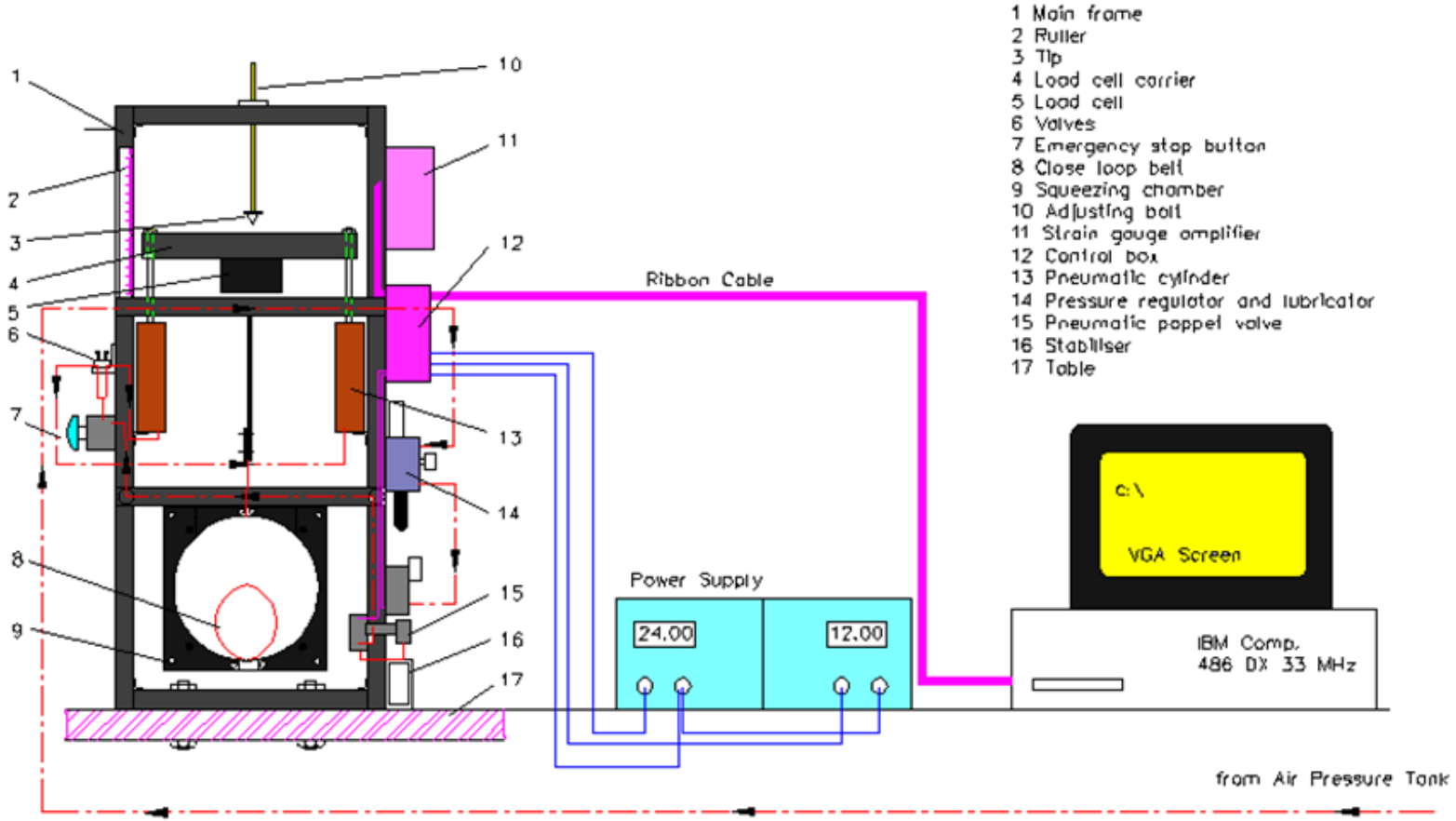


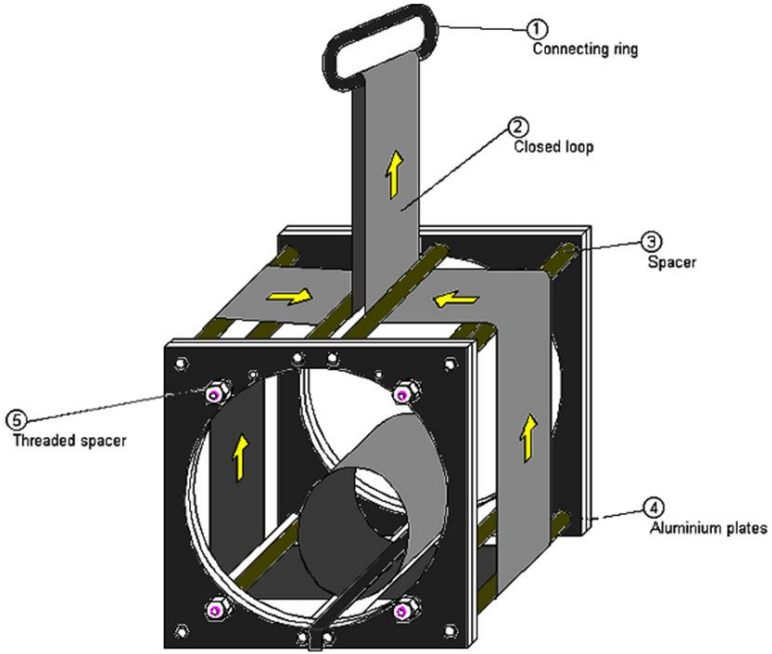


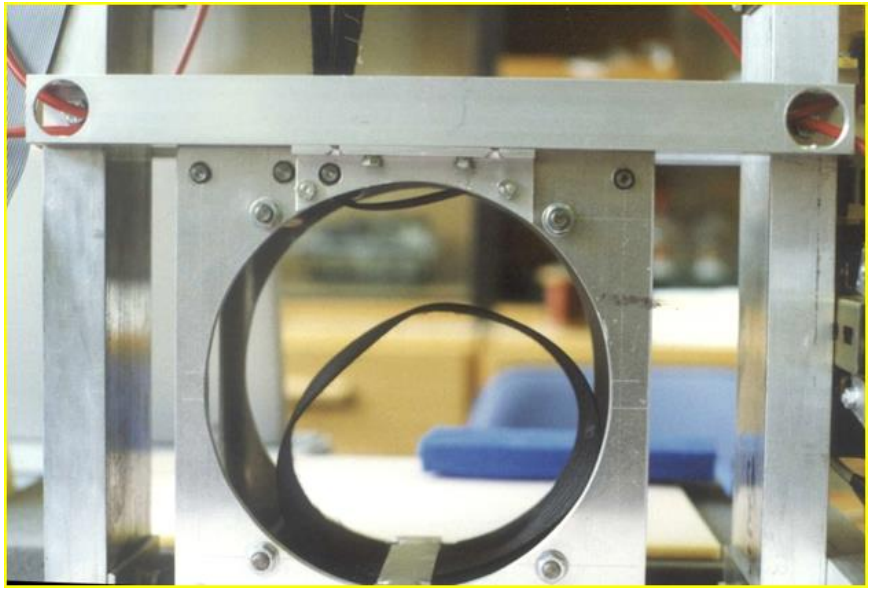
A total of 814 actuators are modeled

PHYSICS in COMPUTER ANIMATIONS and GAMES

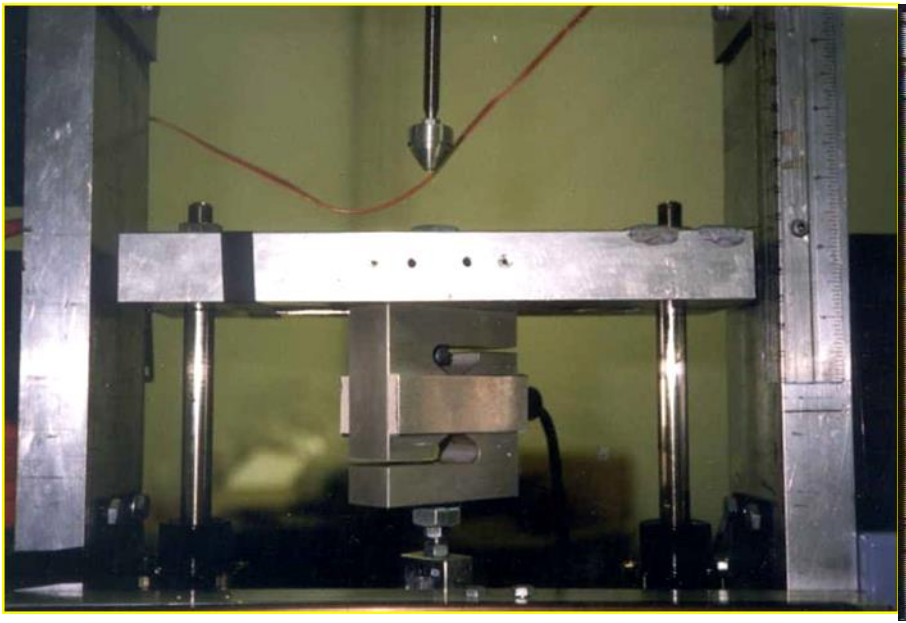


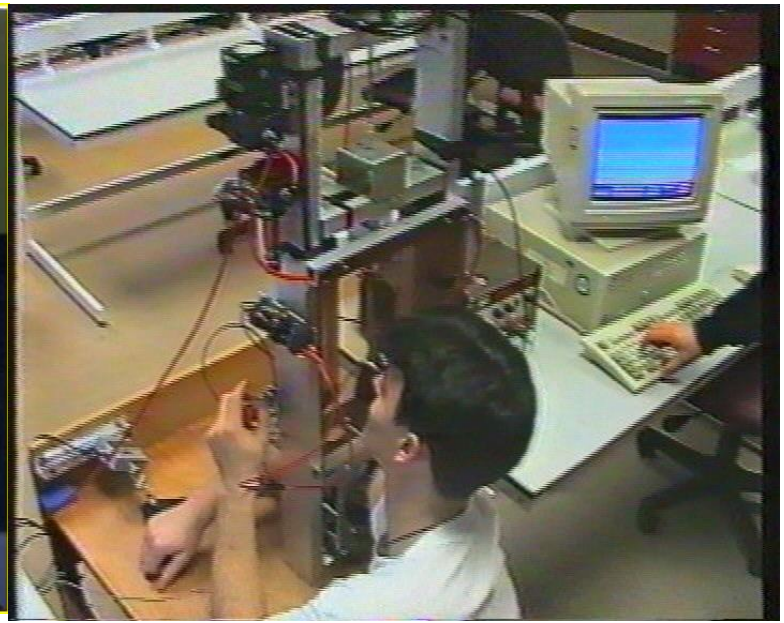

1 Main frame
2 Ruller
3 Tip
4 Load cell carrier
5 Load cell
6 Valves
7 Emergency stop button
8 Close loop belt
9 Squeezing chamber
10 Adjusting bolt
11 Strain gauge amplifier
12 Control box
13 Pneumatic cylinder
14 Pressure regulator and lubricator
15 Pneumatic poppet valve
16 Stabiliser
17 Table
Ribbon Cable
Power Supply
24.00
12.00
C:\
VGA Screen
IBM Comp.
486 DX 33 MHz
from Air Pressure Tank


1 Connecting ring
2 Closed loop
3 Spacer
4 Aluminium plates
5 Threaded spacer

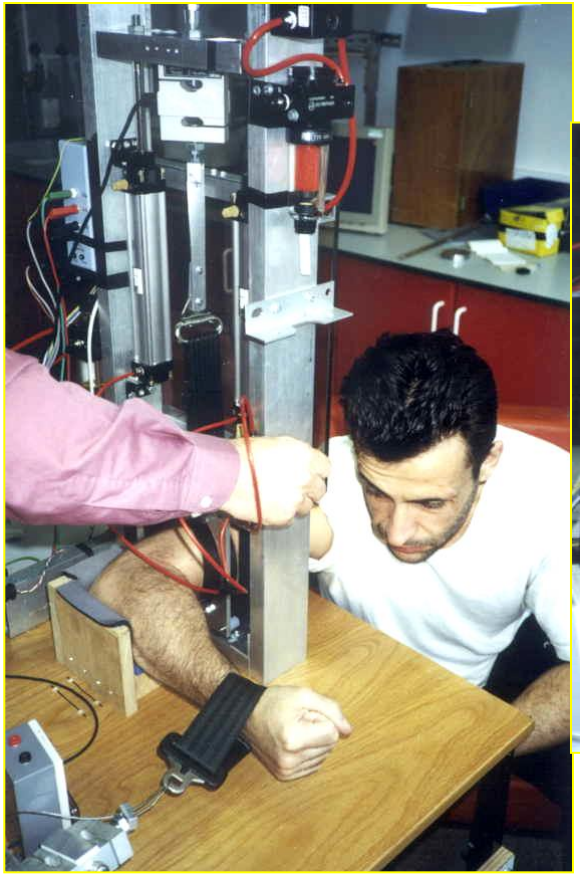


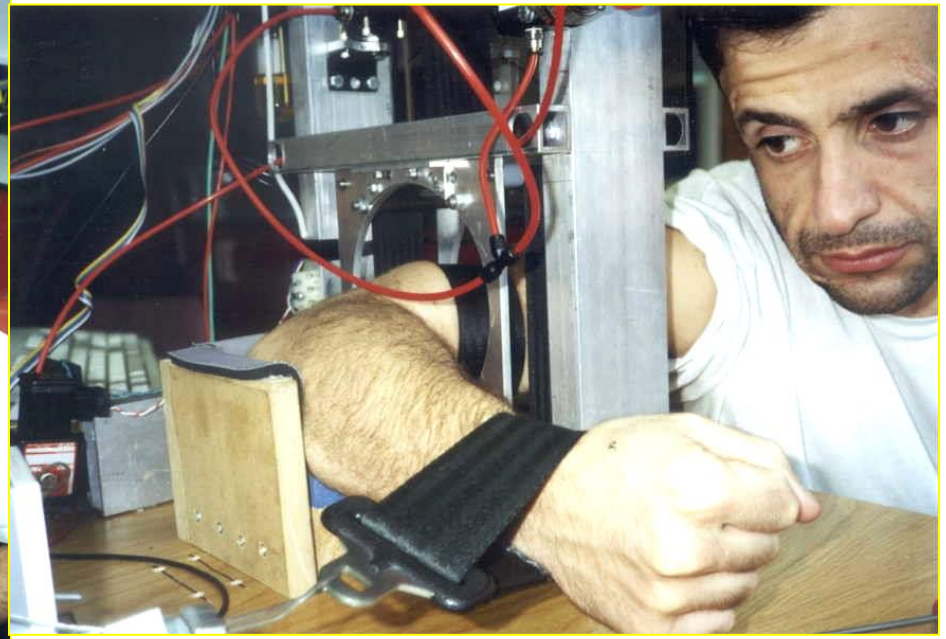
PHYSICS in COMPUTER ANIMATIONS and GAMES

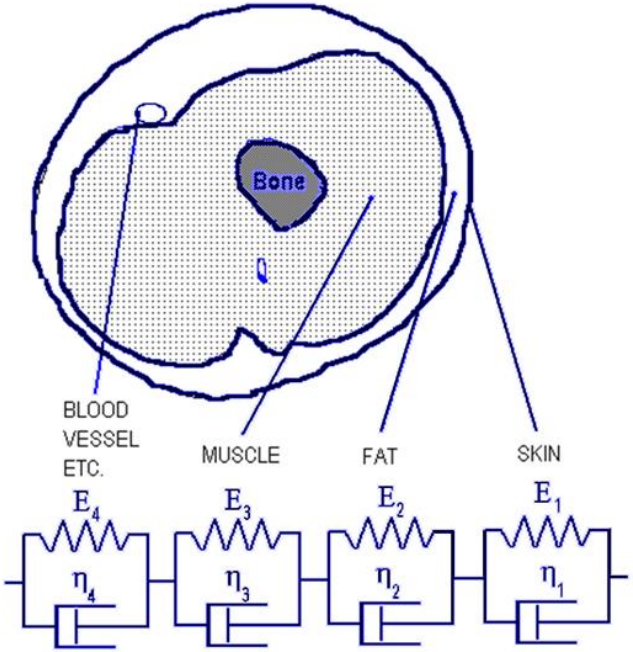




PHYSICS in COMPUTER ANIMATIONS and GAMES







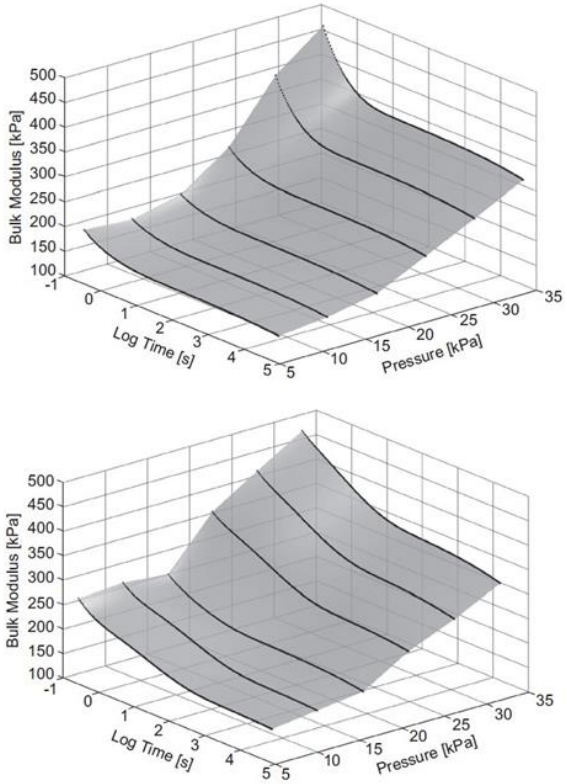


Fig. 10. Bulk properties of muscular bulk tissue in contracted condition.

S. Arıtan et al. *'A mechanical model representation of the in vivo creep behaviour of muscular bulk tissue'* / **Journal of Biomechanics**

Finite Element Modelling

•ELEMENT MESHES CAN BE COMPLICATED

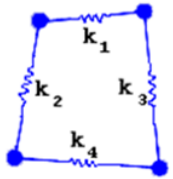
2D Element

$k_1 = k_2 = k_3 = k_4$

isotropic

$k_1 \neq k_2 \neq k_3 \neq k_4$

anisotropic



3D Element

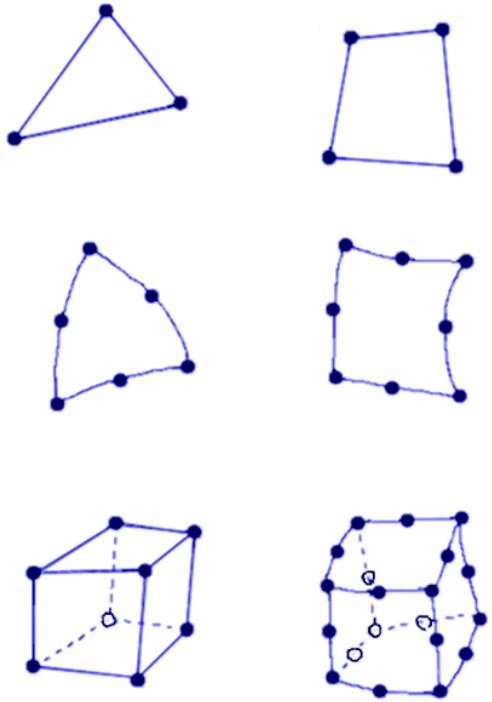
$$\{f\} = [k]\{u\}$$

f : force vector

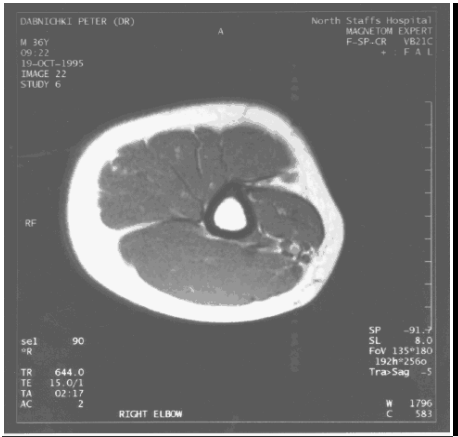
k : stifness matrix

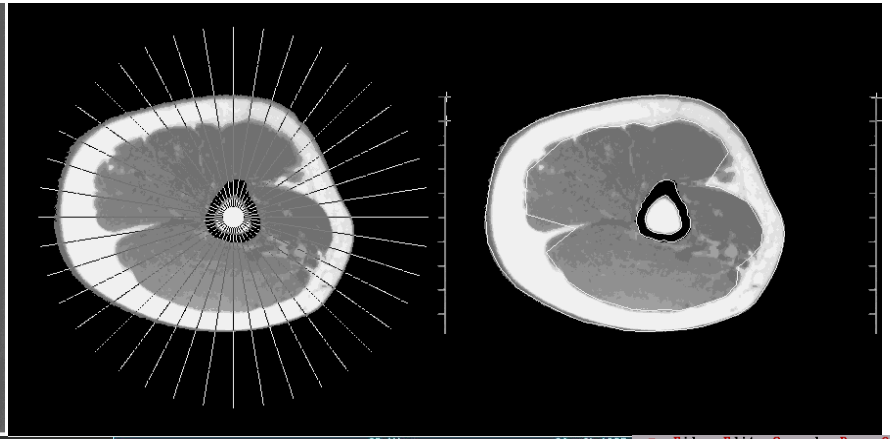
u : displacement vector

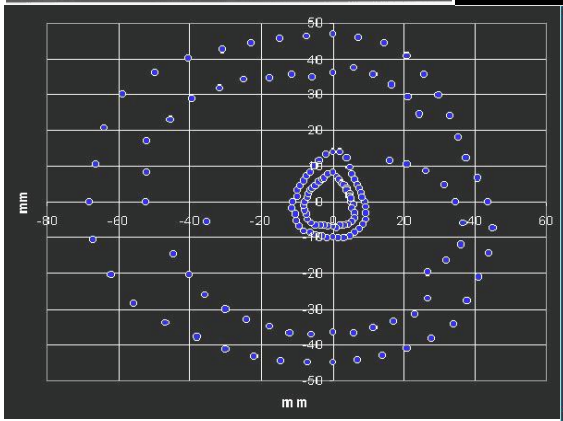
Geometry of Elements

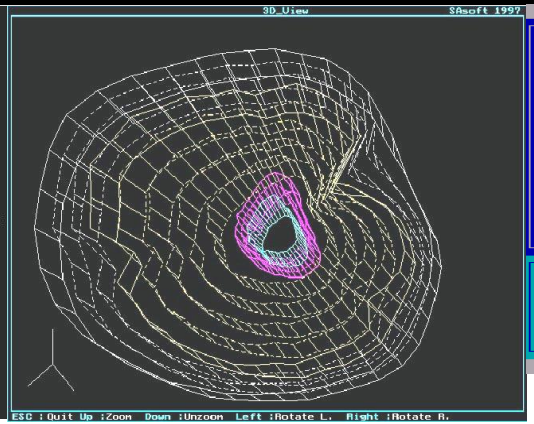


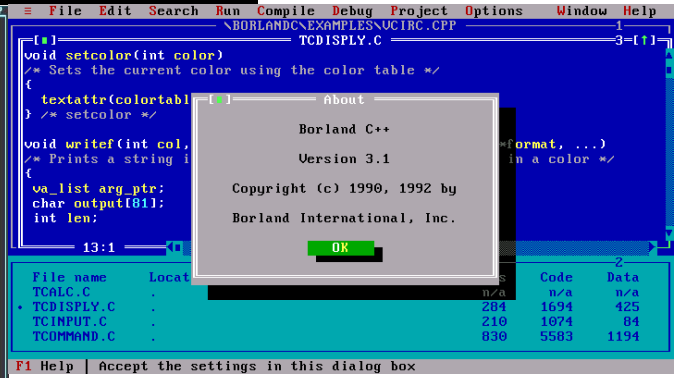
PHYSICS in COMPUTER ANIMATIONS and GAMES





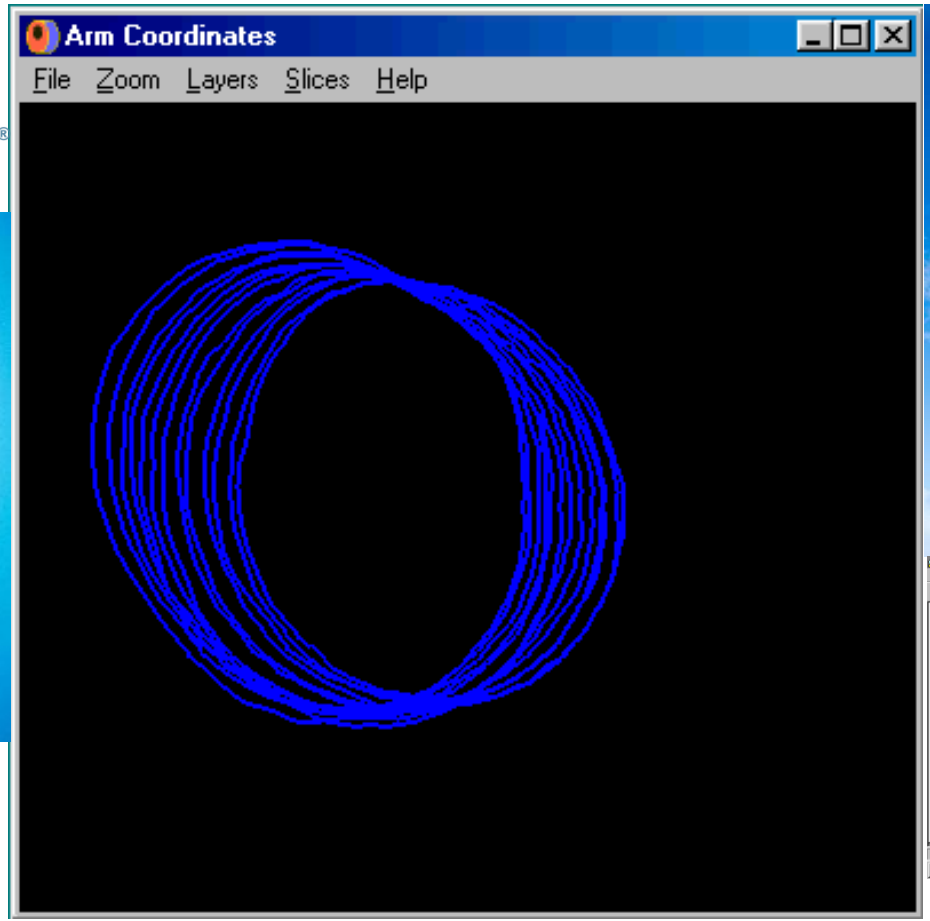


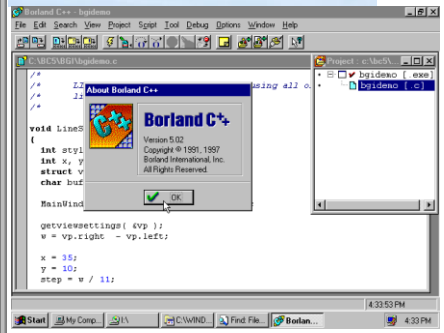




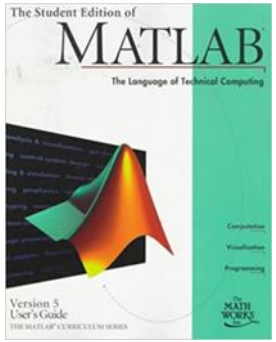
Arıtan S. *et al*. (1997) Program for generation of three-dimensional finite element mesh from magnetic imaging scans. **Medical Enginering & Physics**.

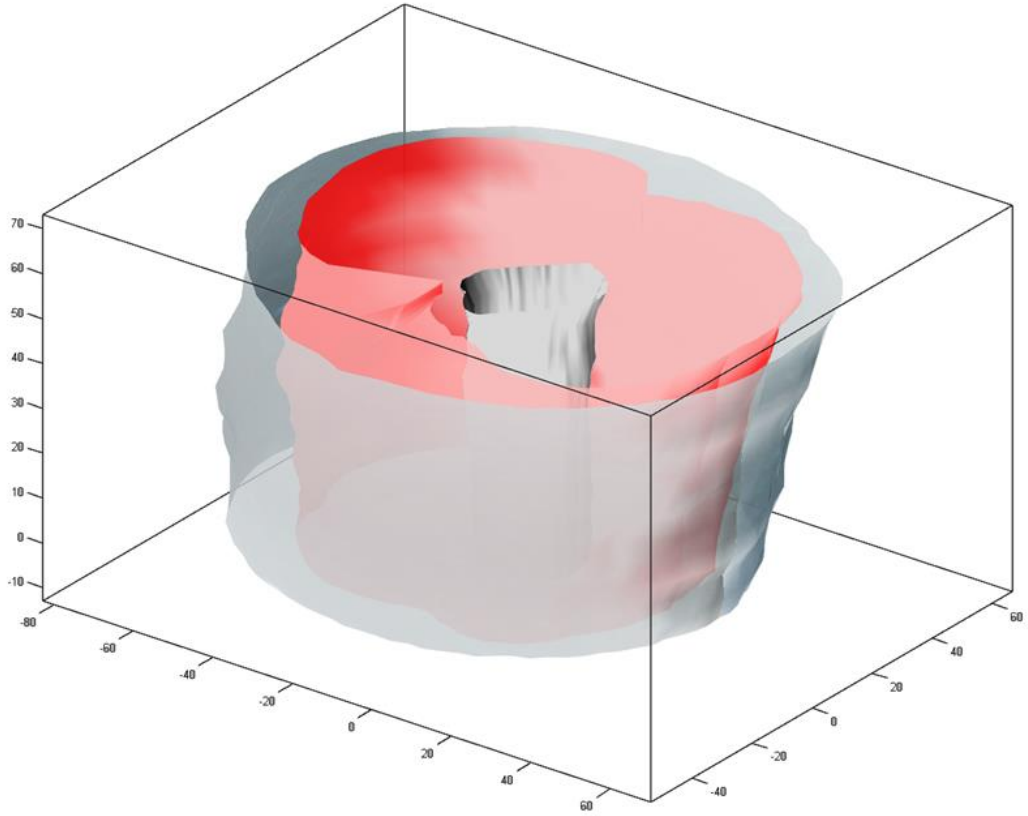
PHYSICS in COMPUTER ANIMATIONS and GAMES



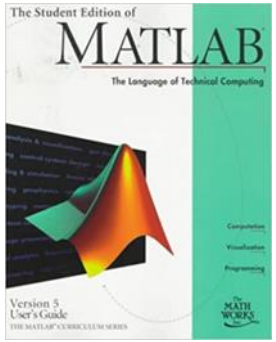


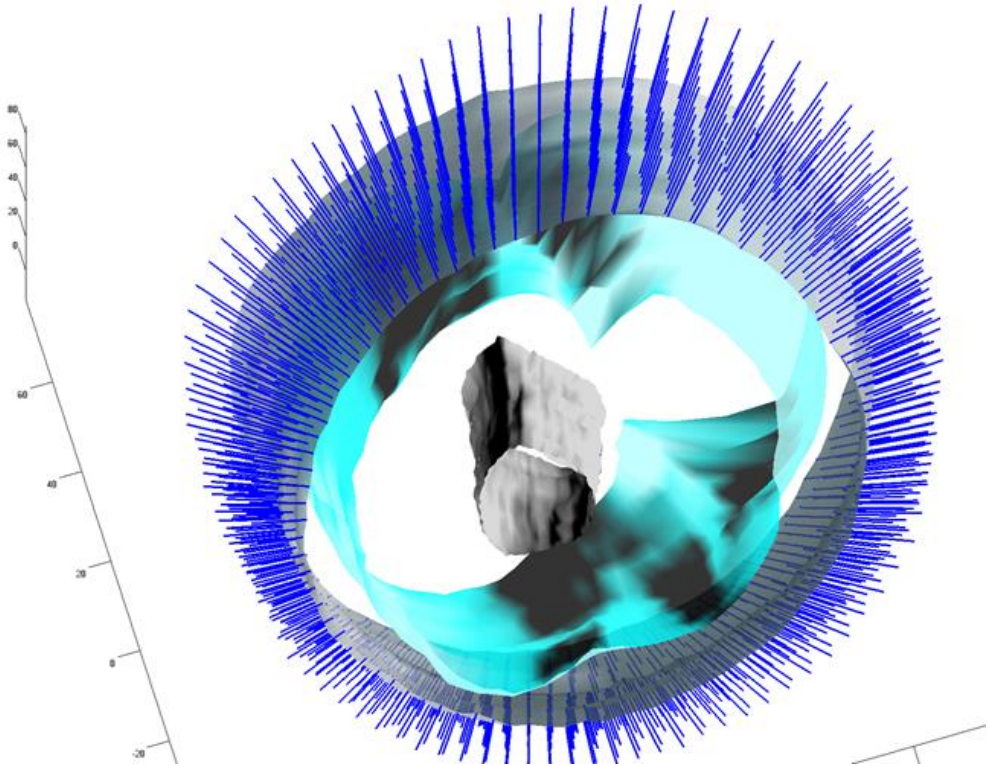
PHYSICS in COMPUTER ANIMATIONS and GAMES





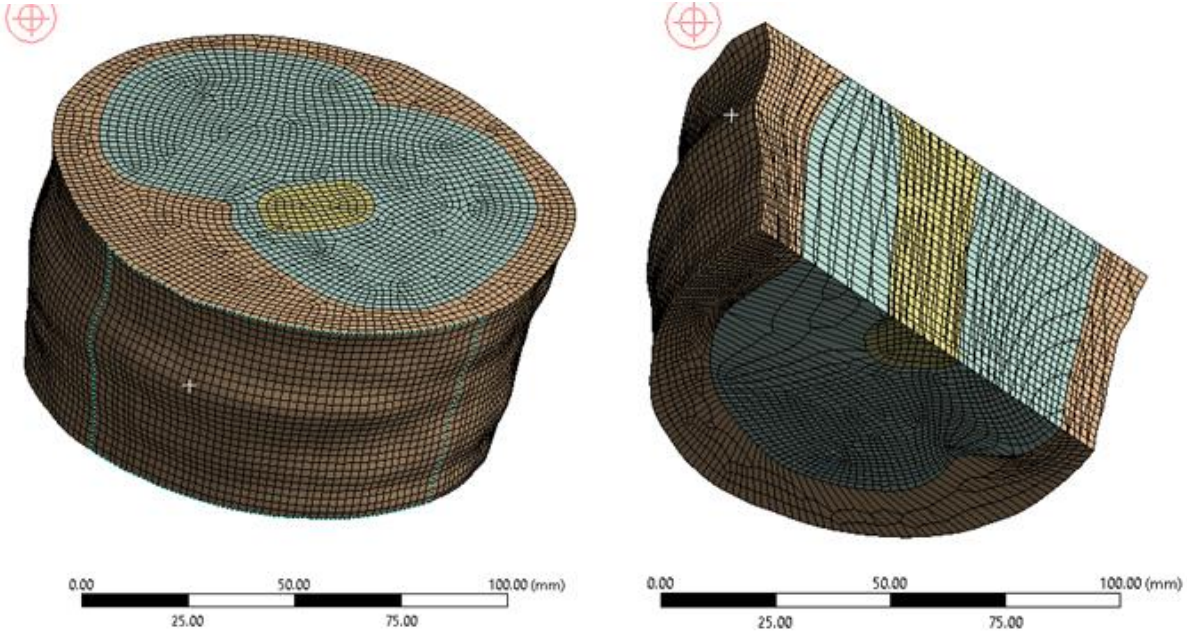
PHYSICS in COMPUTER ANIMATIONS and GAMES





PHYSICS in COMPUTER ANIMATIONS and GAMES

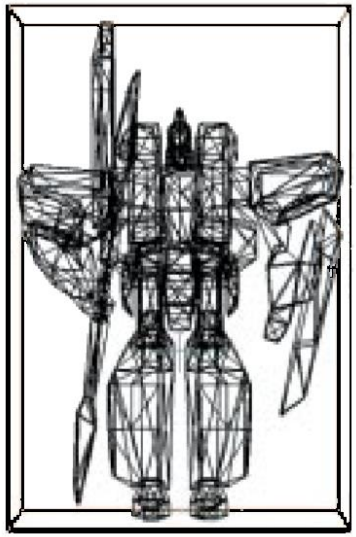
The SGI Origin 2000 is a family of mid-range and high-end server computers developed and manufactured by Silicon Graphics (SGI). They were introduced in 1996 to succeed the SGI Challenge and POWER Challenge.

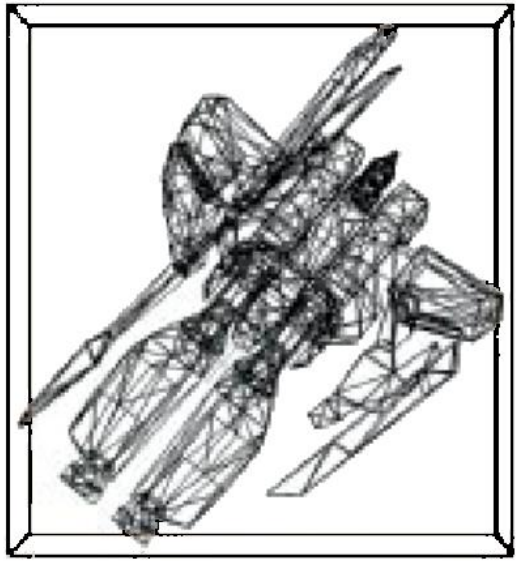


ABAQUS is a software suite for finite element analysis and computer-aided engineering, originally released in 1978.

Axis-Aligned minimum Bounding Box (AABB)

An axis-aligned bounding box (AABB) is also a very quick way of determining collisions. The fit is generally better than a bounding sphere (especially if the object you are bounding is a box itself).

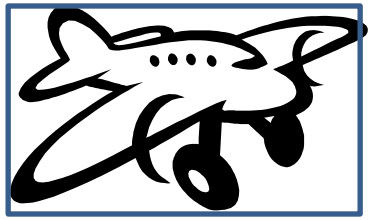




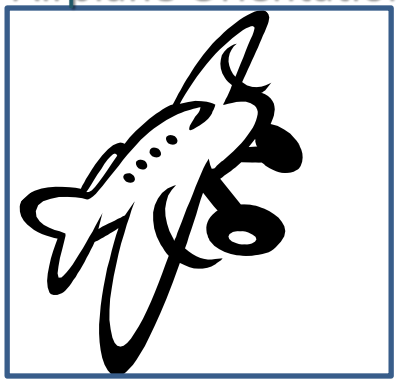
Axis-Aligned minimum Bounding Box (AABB)

- A box that is
  Defined by the min and max coordinates of an object
  Always aligned with the coordinate axes

Initial Airplane Orientation



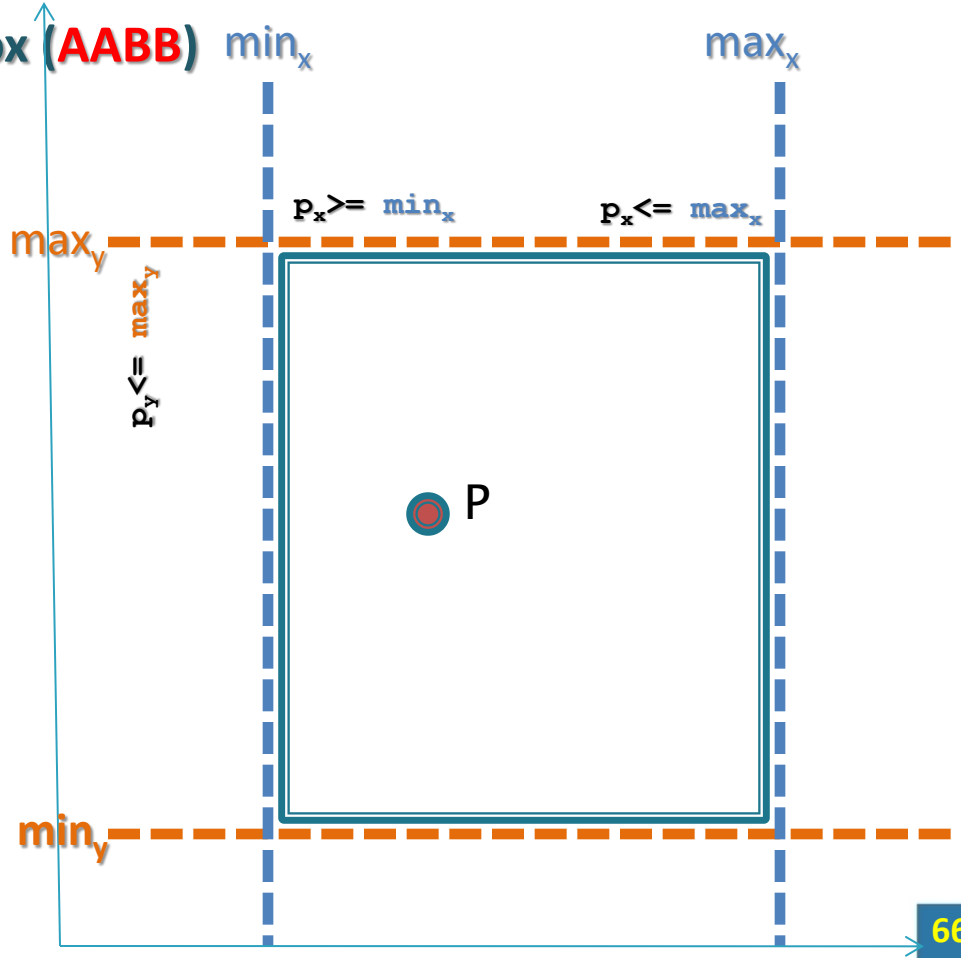
Airplane Orientation 2



Axis-Aligned minimum Bounding Box (AABB)

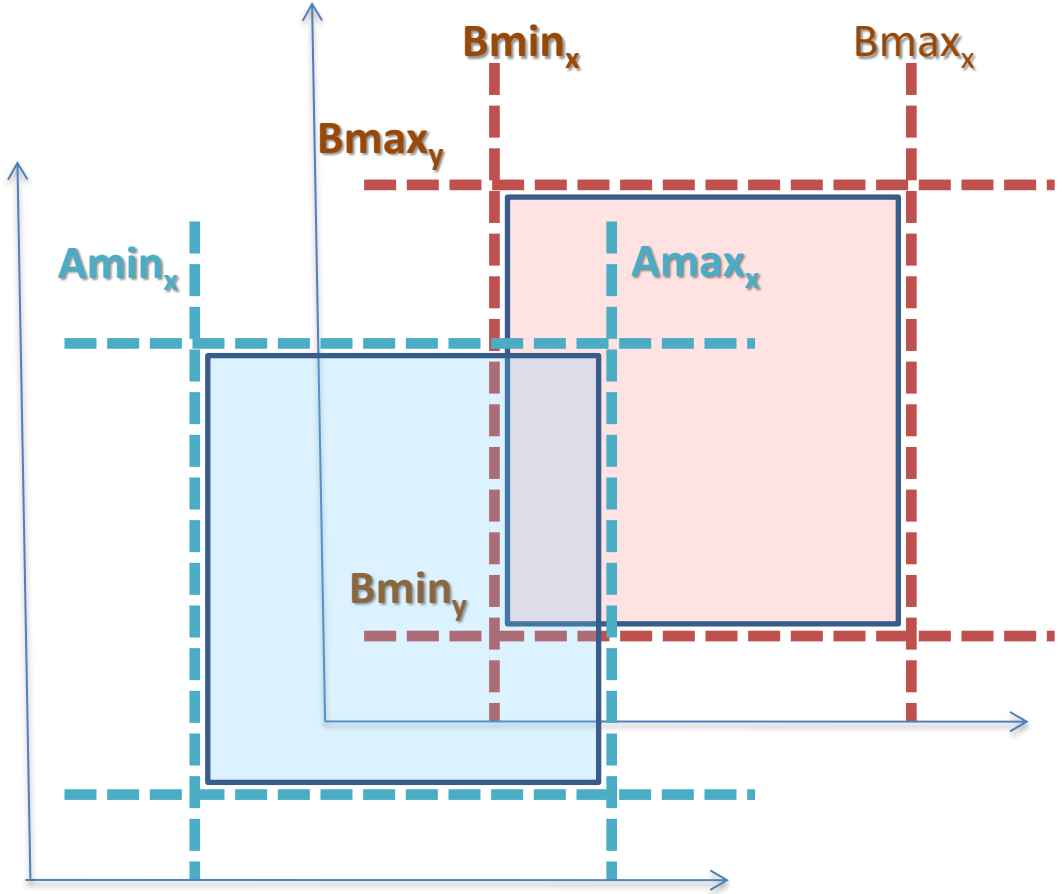
- How can we tell if a point p is inside the box?
  - Do a bunch of ifs
    - if

```
px<= maxx and
py<= maxy and
px>= minx and
py>= miny :
    collide = True
elif
    collide = False
```



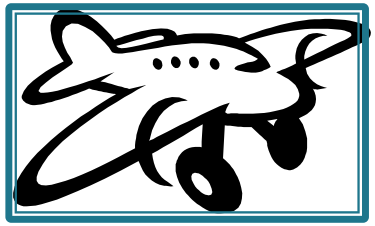
Comparing AABBs

```
if mins/maxes overlap:
    collide = True
else:
    collide = False
```

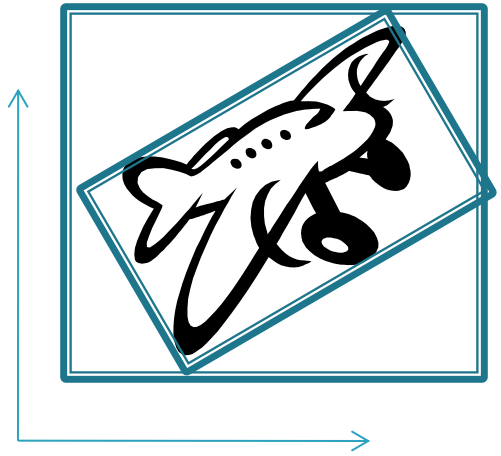


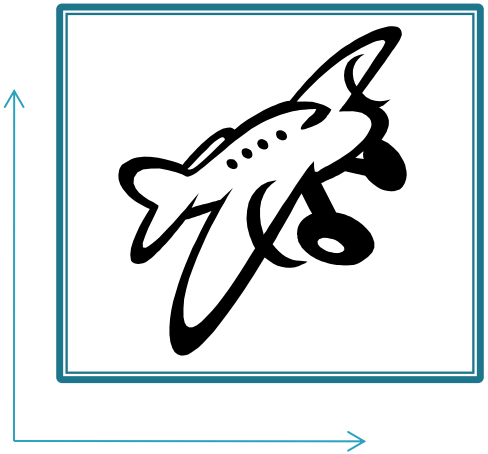
AABB Approach

Initialization: Iterate through vertices and find mins and maxes



After Transformations: Iterate through AABB vertices and find mins and maxes





AABB Approach

- Initialization
  iterate through all vertices of your model to find the mins and maxes for x, y, and z
- During runtime
  Test if any of the AABB mins/maxes of one object overlap with another object's AABB mins/maxes
    MAKE SURE THAT THE AABB VALUES ARE IN THE SAME COORDINATE FRAME (e.g., world coordinates)!


```
import pygame
import random

red = [255, 0, 0]
green = [0, 255, 0]
blue = [0, 0, 255]
white = [255, 255, 255]
black = [0, 0, 0]
UP = [0, -1]
DOWN = [0, 1]
LEFT = [-1, 0]
RIGHT = [1, 0]
NOTMOVING = [0, 0]
# constants end
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
# Classes
class collidable:
    x = 0
    y = 0
    w = 0
    h = 0
    rect = pygame.Rect(x, y, w, h)
    color = [0, 0, 0]

    def __init__(self, x, y, w, h, color):
        self.x = x
        self.y = y
        self.w = w
        self.h = h
        self.color = color
        self.rect = pygame.Rect(x, y, w, h)

    def draw(self):
        pygame.draw.rect(screen, self.color, [self.x,self.y,self.w, self.h], 6)
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
class player:
    x = 0
    y = 0
    speed = 0
    rect = pygame.Rect(x, y, 20, 20)

    def __init__(self, x, y, speed):
        self.x = x
        self.y = y
        self.speed = speed
        self.rect = pygame.Rect(self.x, self.y, 20, 20)
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
  def draw(self):
    if player_moving==LEFT:
      pygame.draw.polygon(screen,black,[(self.x-10,self.y),(self.x+10,self.y-
10),(self.x+10,self.y+10)])
    elif player_moving==RIGHT:
      pygame.draw.polygon(screen,black,[(self.x+10,self.y),(self.x-10,self.y-
10),(self.x-10,self.y+10)])
    elif player_moving==UP:
      pygame.draw.polygon(screen,black,[(self.x,self.y-10),(self.x+10,
self.y+10),(self.x-10,self.y+10)])
    elif player_moving==DOWN:
      pygame.draw.polygon(screen,black,[(self.x,self.y+10),(self.x+10,self.y-
10),(self.x-10,self.y-10)])
    else:
      pygame.draw.rect(screen,black,pygame.Rect(self.x-10,self.y-10,20,20), 6)
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
    def setpos(self, x, y):
        self.x = x
        self.y = y

    def move(self, direction):
        self.x = self.x + direction[0]*self.speed
        self.y = self.y + direction[1]*self.speed
        self.rect = pygame.Rect(self.x, self.y, 20, 20)
# Classes End
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
# globals
pygame.init()
screenSize = [800, 600]
screenBGColor = white
screen = pygame.display.set_mode(screenSize)
pygame.display.set_caption("Move the Block")
player = player(screenSize[0]/2, screenSize[1]/2, 9)
collidables = []
clock = pygame.time.Clock()
for i in range(10):
    collidables.append(collidable(random.randrange(0, screenSize[0]),
        random.randrange(0, screenSize[1]), random.randrange(10, 200),
        random.randrange(10, 200), blue))

running = True
# globals end
player_moving = NOTMOVING
```

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
# Functions
def render():
    screen.fill(screenBGColor)
    clock.tick(60)
    player.draw()
    for c in collidables:
        c.draw()
    pygame.display.flip()


def tick(player_moving):
    for c in collidables:
        if player.rect.colliderect(c.rect):
            player_moving = NOTMOVING
            print("hit"+str(c.rect)+" with "+str(player.rect))
    player.move(player_moving)

# Functions End
```

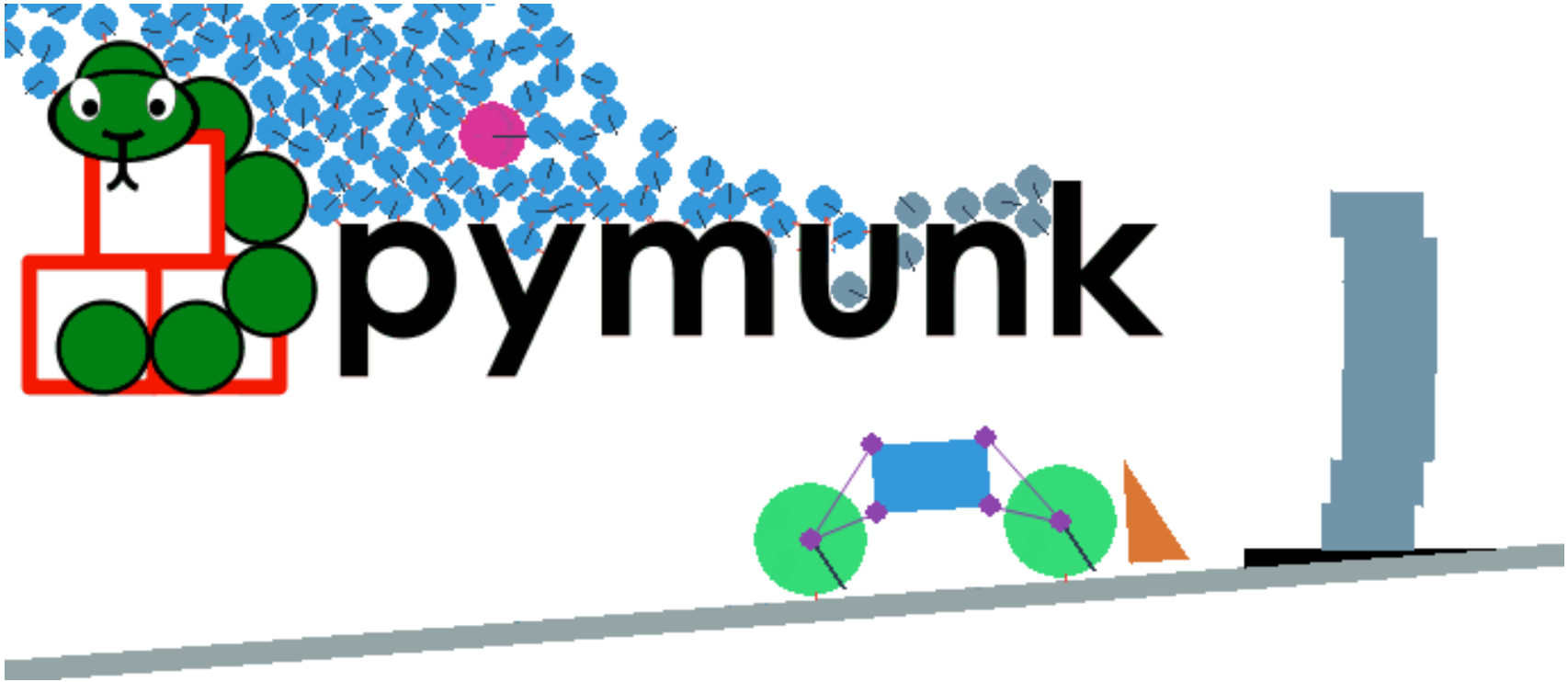
```
# main loop
while running:
  for event in pygame.event.get():
     if event.type == pygame.QUIT:
      running = False
     if event.type == pygame.KEYDOWN:
       if event.key == pygame.K_LEFT:
         player_moving = LEFT
       if event.key == pygame.K_RIGHT:
         player_moving = RIGHT
       if event.key == pygame.K_UP:
         player_moving = UP
       if event.key == pygame.K_DOWN:
         player_moving = DOWN
     else:
       player_moving = NOTMOVING
  tick(player_moving)
  render()
# main loop end
pygame.quit()
```

Gilbert–Johnson–Keerthi distance algorithm

GJK distance algorithm is a method of determining the minimum distance between two convex sets.

GJK algorithms are often used incrementally in simulation systems and **video games**. In this mode, the final simplex from a previous solution is used as the initial guess in the next iteration, or "frame". If the positions in the new frame are close to those in the old frame, the algorithm will converge in one or two iterations. This yields collision detection systems which operate in near-constant time.

The algorithm's stability, speed, and small storage footprint make it popular for **realtime** collision detection, especially in physics engines for video games.


pymunk

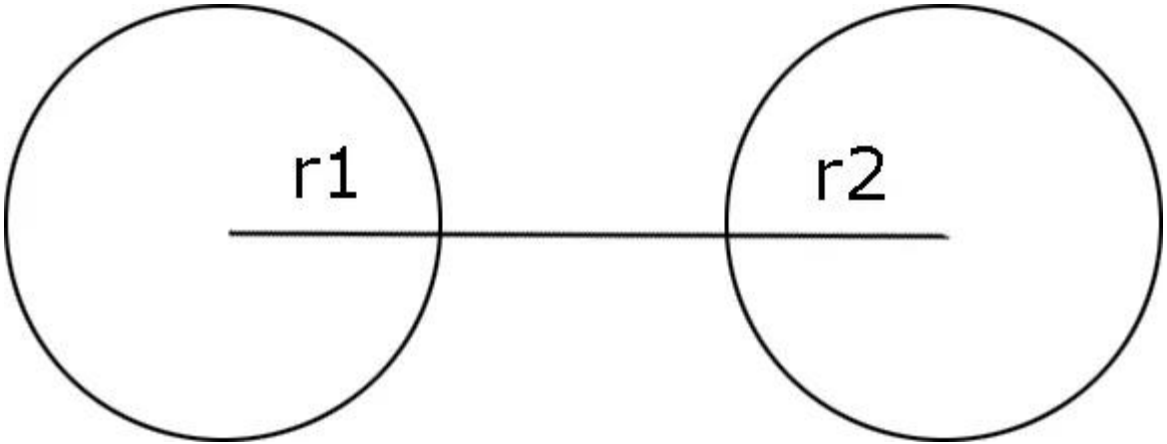
Collision Detection Algorithm

Just as the impulse solver, the collision detection is also handled by the underlying C-library Chipmunk2D.

Chipmunk uses GJK/EPA to find collisions between the tricky cases (e.g. polygons, segment shapes). There is a blog post here with more details.

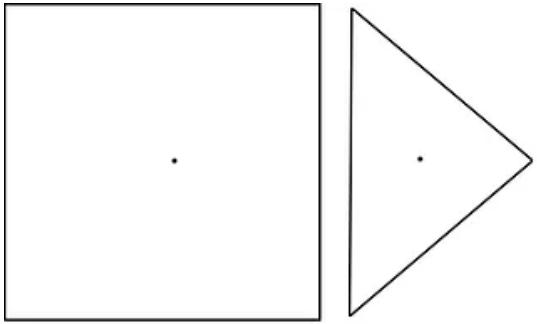
pymunk GJK (Gilbert–Johnson–Keerthi)

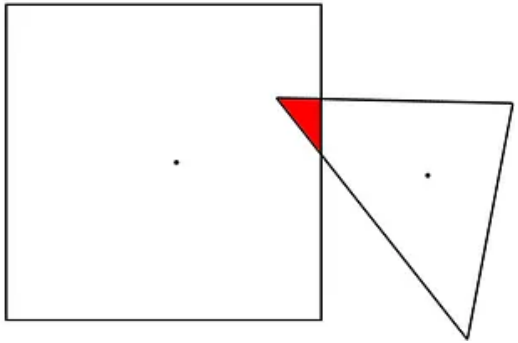
The simplest method to detect a collision in 2D space is to treat all objects as circles (Gilbert, Johnson and Keerthi, 1988:193); if the sum of the circles' radii is greater than or equal to the difference between their centres, then the circles must be touching or overlapping and a collision would be detected.



pymunk GJK (Gilbert–Johnson–Keerthi)

This idea can be generalised to 3D space by treating objects as spheres, as well as to any higher dimensions. However, it is unrealistic to easily simplify every shape into a circle. One such widely-used method is the Gilbert-Johnson-Keerthi (GJK) algorithm by Gilbert, Johnson and Keerthi (1988). To develop an understanding of how this algorithm works, we'll first go over some fundamental concepts.

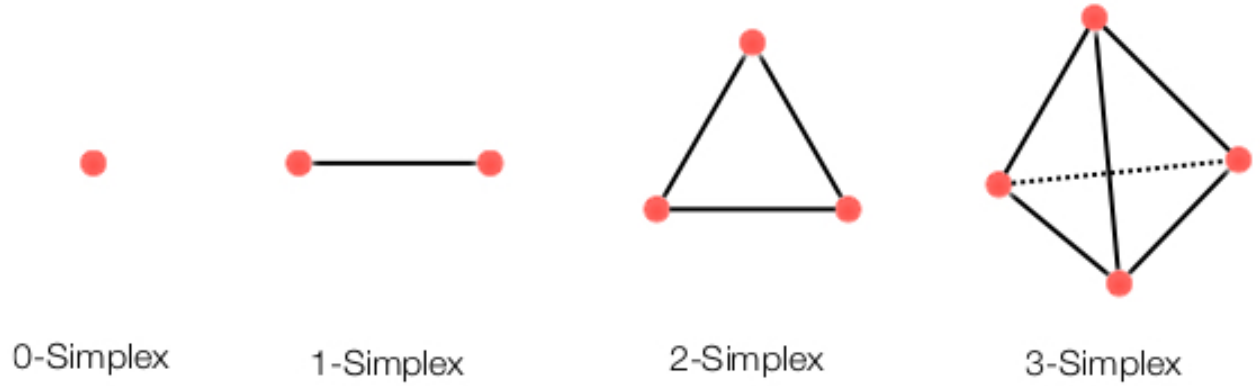




pymunk

Simplex

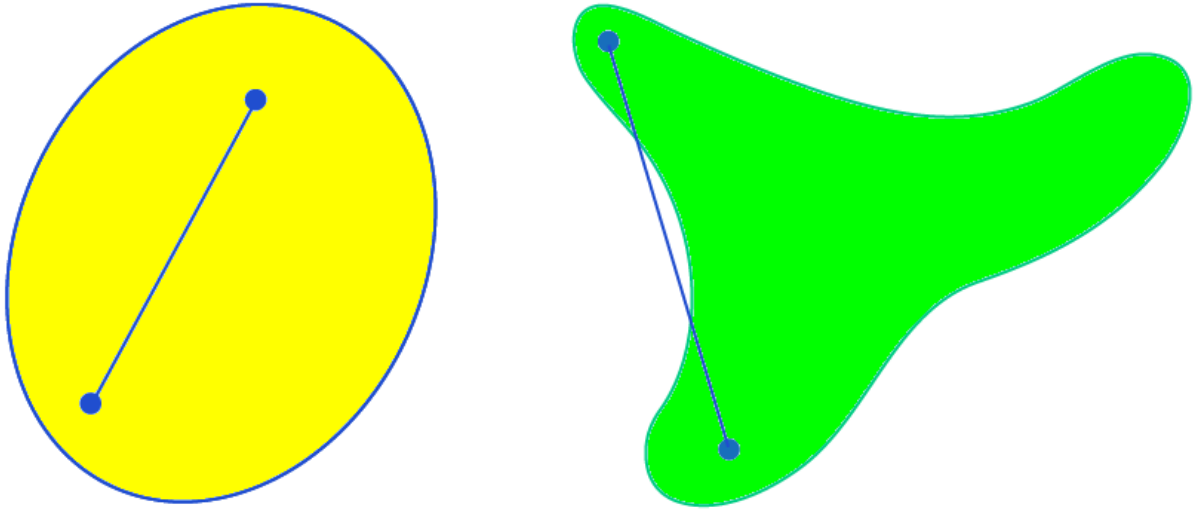
In collision detection, the term Simplex is used a lot. A Simplex refers to either a point, line segment, triangle or tetrahedron. For example, the 0-simplex is a point, the 1-simplex is a line segment, the 2-simplex is a triangle, and the 3-simplex is a tetrahedron.



pymunk

Convex and Concave Shapes

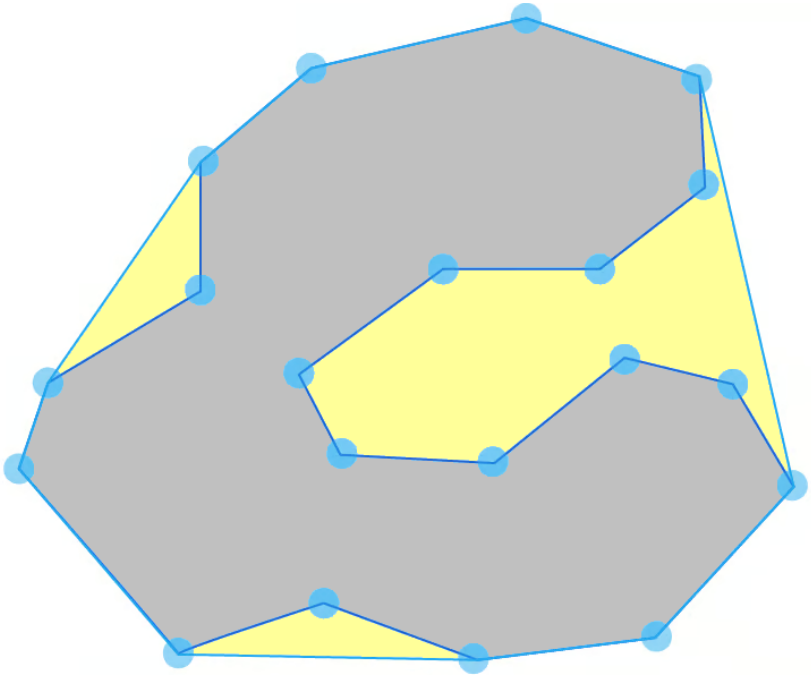
In a convex shape, a line segment between any two points within the shape always falls completely inside the shape.



pymunk

Convex Hull

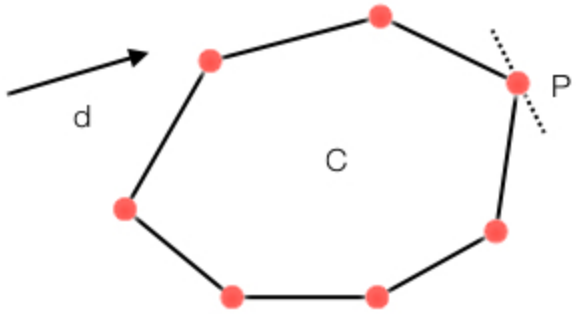
The convex hull of a shape is the smallest convex shape that fully contains it.



pymunk

Supporting Point

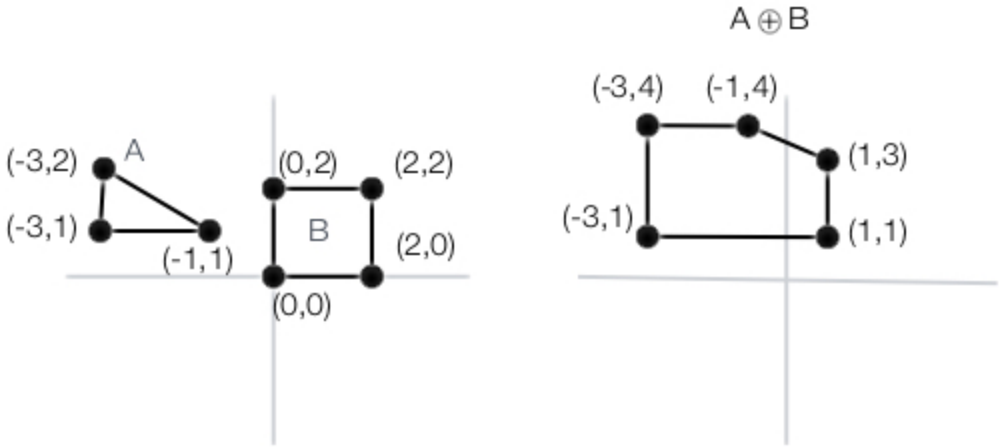
In a convex object, the supporting point is the most distant point in a given direction. In some books, they are referred as extreme points. In the illustration below, the supporting point in direction d is P.



pymunk

Minkowski Sum

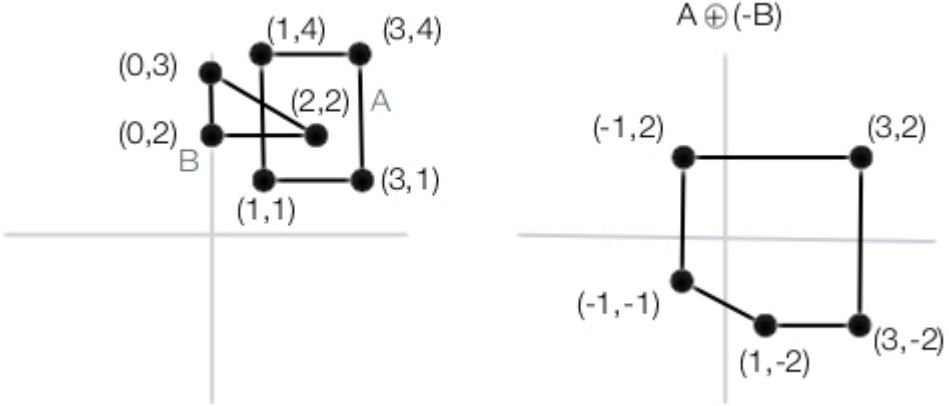
In collision detection, there are two important operations which you need to understand. They are the Minkowski Sum and the Minkowski Difference. Visually, the Minkowski **sum** can be seen as the region swept by Object A translated to every point in Object B.



pymunk

Minkowski Difference

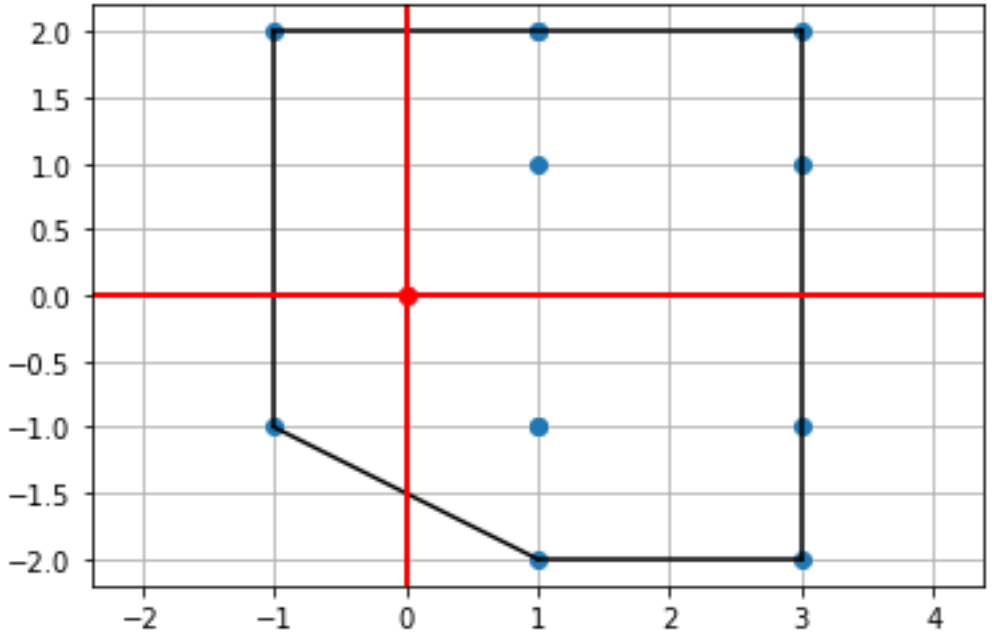
The Minkowski difference is the region swept by Object A translated to every point negated in Object B. The Minkowski difference is a significant operation in collision detection because two objects A and B collide if their Minkowski difference contains **the origin**. The GJK algorithm uses this fact to determine if two convex objects have collided.



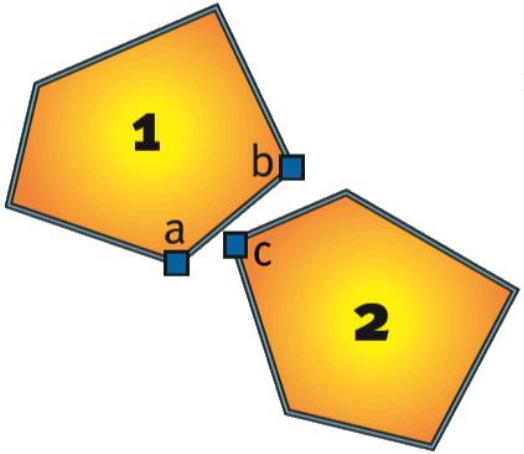
PHYSICS in COMPUTER ANIMATIONS and GAMES

```
from scipy.spatial import ConvexHull
import numpy as np
import matplotlib.pyplot as plt

A = [(1,4),(1,1),(3,1),(3,4)]
B = [(0,3),(0,2),(2,2)]

points = np.asarray([(xA-xB, yA-yB) for xA, yA in A for xB, yB in B])
hull = ConvexHull(points)

plt.plot(points[:,0], points[:,1], 'o')
plt.plot(0, 0, 'ro')

for simplex in hull.simplices:
    plt.plot(points[simplex, 0], points[simplex, 1], 'k-')

plt.plot(0, 0, 'ro')
plt.axhline(linewidth=2, color='r')
plt.axvline(linewidth=2, color='r')

plt.axis('equal')
plt.grid()
```

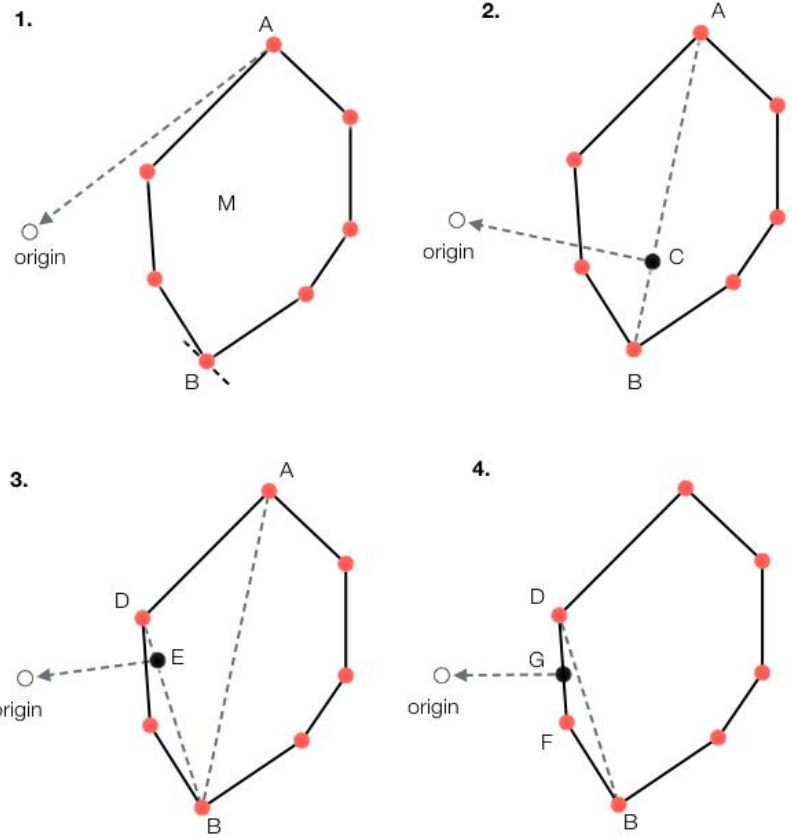
Minkowski Difference



Inside the Polygon




```
# determine if a point is inside a given polygon or not
# Polygon is a list of (x,y) pairs.

def point_inside_polygon(x,y,poly):

    n = len(poly)
    inside =False

    p1x,p1y = poly[0]
    for i in range(n+1):
        p2x,p2y = poly[i % n]
        if y > min(p1y,p2y):
            if y <= max(p1y,p2y):
                if x <= max(p1x,p2x):
                    if p1y != p2y:
                        xinters = (y-p1y)*(p2x-p1x)/(p2y-p1y)+p1x
                    if p1x == p2x or x <= xinters:
                        inside = not inside
        p1x,p1y = p2x,p2y

    return inside
```



1. The algorithm arbitrarily starts with the vertex A as the initial simplex in set Q, Q={A}.
2. Searching for the supporting point in direction -A results in B. B is added to the Simplex set, Q={A, B}
3. The point in the convex hull Q closest to the origin is C. Because both A and B are needed to express C as a convex combination, both are kept in Q.
4. D is the supporting point in direction -C and it is added to Q, giving Q={A, B, D}.
5. The closest point in the convex hull Q closest to the origin is now E.
6. Because only B and D are needed to express E as a convex combination of vertices in Q, Q is updated to Q={B, D}. The supporting point in direction -E is F, which is added to Q.
7. The point on the convex hull Q closest to the origin is now G.
8. D and F are the smallest set of vertices in Q need to express G as a convex combination. Q is updated to Q={D, F}.
9. At this point, because no vertex is closer to the origin in direction -G than G itself, G must be the closest point to the origin, and the algorithm terminates. **No collision occurred**.

https://winter.dev/articles/gjk-algorithm/app/demo.html