```
Tony Florida
EN 605.713 Robotics
Johns Hopkins University
Midterm Project
Given: 29 Jan 2014
Due: 12 Mar 2014
Written Documentation
```
View as PDF

Notes/Assumptions

- Vehicle and wheels are drawn to scale.
- Vehicle coordinate system aligns with global coordinate system upon page load.
- Each cell in the grid is 6" x 6".
- Environment is flat, high friction, infinite area.
- Max speed of vehicle is 15 ft/sec.
- Angles are specified clockwise from the positive Y axis.

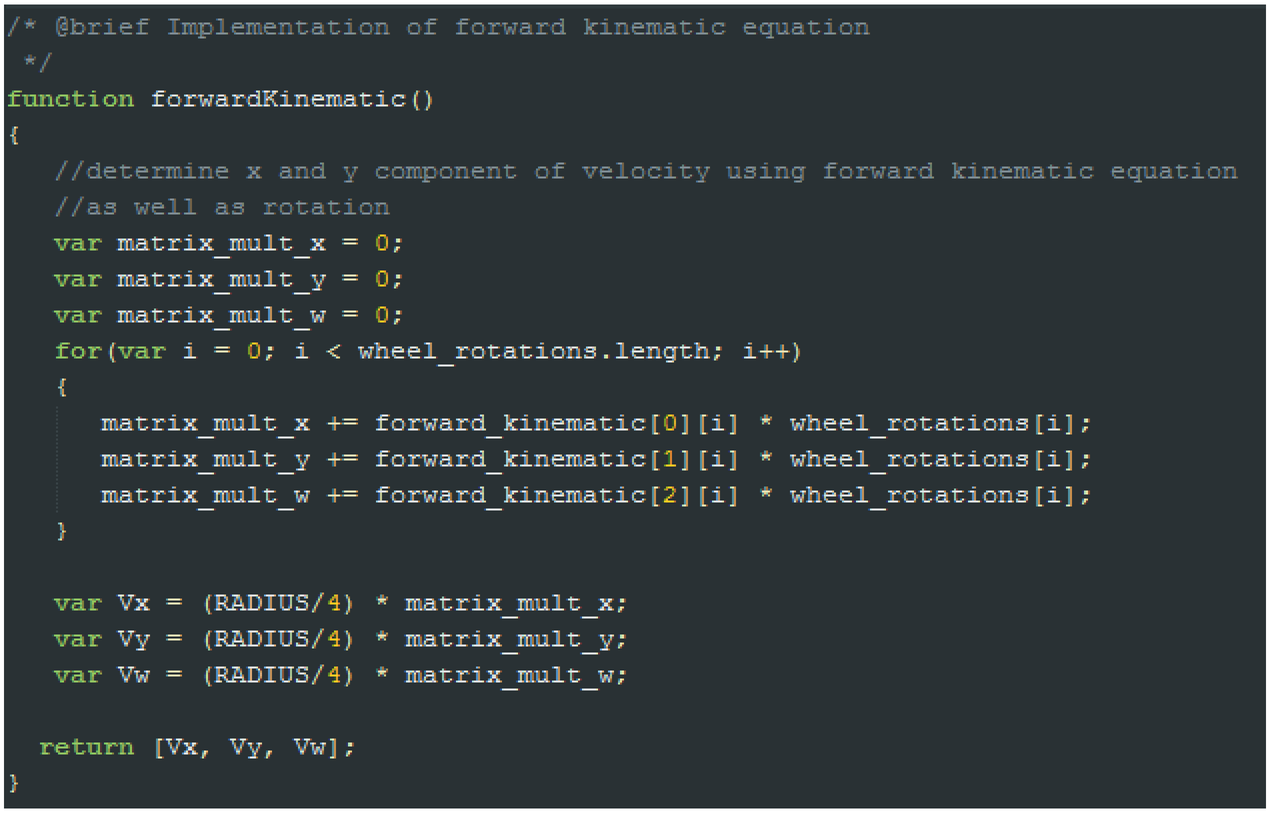
Kinematic Equations

- Forward Kinematic Equations

Equation

$$\begin{bmatrix} V_X \\ V_Y \\ \omega \end{bmatrix} = \frac{R}{4} \begin{bmatrix} 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 \\ -\frac{1}{(L+H)} & \frac{1}{(L+H)} & -\frac{1}{(L+H)} & \frac{1}{(L+H)} \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \psi_4 \end{bmatrix}$$

Implementation

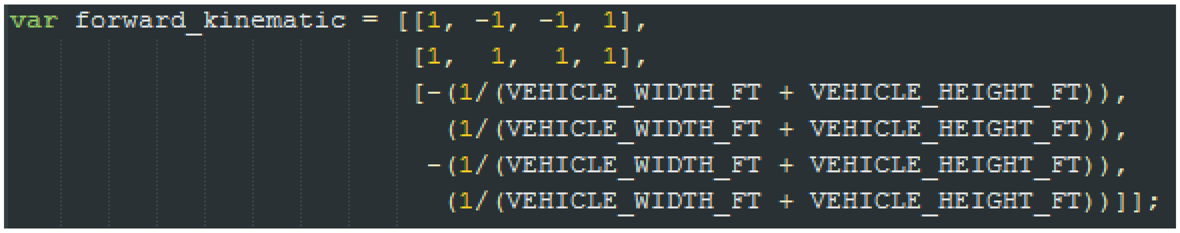


```
/* @brief Implementation of forward kinematic equation
 */
function forwardKinematic()
{
   //determine x and y component of velocity using forward kinematic equation
   //as well as rotation
   var matrix_mult_x = 0;
   var matrix_mult_y = 0;
   var matrix_mult_w = 0;
   for(var i = 0; i < wheel_rotations.length; i++)
   {
      matrix_mult_x += forward_kinematic[0][i] * wheel_rotations[i];
      matrix_mult_y += forward_kinematic[1][i] * wheel_rotations[i];
      matrix_mult_w += forward_kinematic[2][i] * wheel_rotations[i];
   }

   var Vx = (RADIUS/4) * matrix_mult_x;
   var Vy = (RADIUS/4) * matrix_mult_y;
   var Vw = (RADIUS/4) * matrix_mult_w;

  return [Vx, Vy, Vw];
}
```

Code



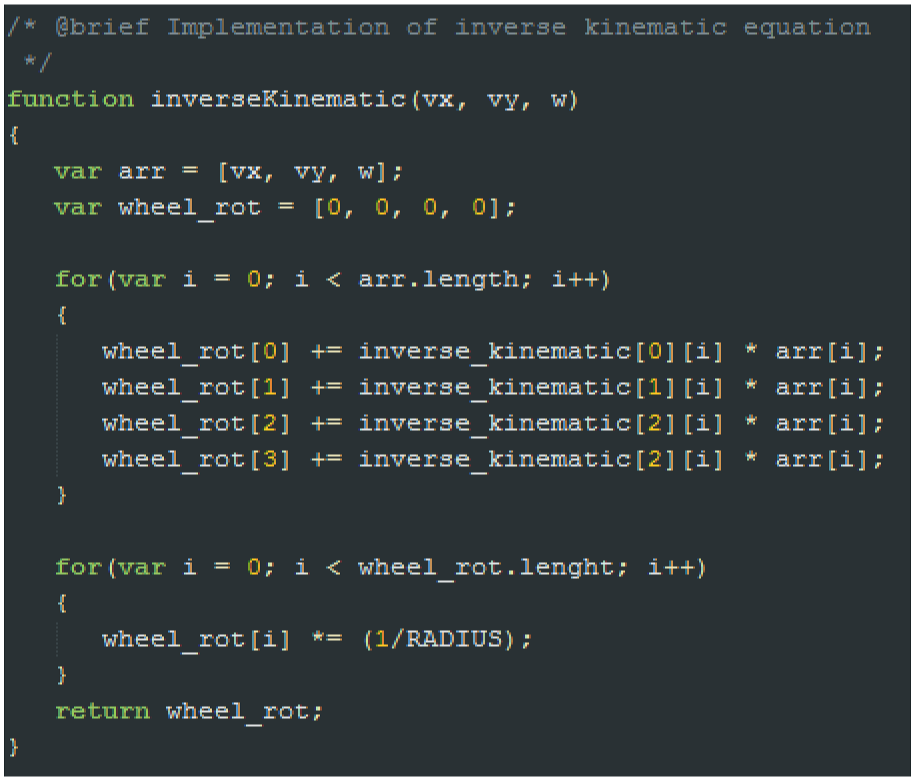
```
var forward_kinematic = [[1, -1, -1, 1],
                         [1,  1,  1, 1],
                         [-(1/(VEHICLE_WIDTH_FT + VEHICLE_HEIGHT_FT)),
                           (1/(VEHICLE_WIDTH_FT + VEHICLE_HEIGHT_FT)),
                          -(1/(VEHICLE_WIDTH_FT + VEHICLE_HEIGHT_FT)),
                           (1/(VEHICLE_WIDTH_FT + VEHICLE_HEIGHT_FT))]];
```

- Inverse Kinematic Equations

Equation

$$\Psi = \begin{bmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \psi_4 \end{bmatrix} = \frac{1}{R} \begin{bmatrix} 1 & 1 & -(L+H) \\ -1 & 1 & (L+H) \\ -1 & 1 & -(L+H) \\ 1 & 1 & (L+H) \end{bmatrix} \begin{bmatrix} V_X \\ V_Y \\ \omega \end{bmatrix}$$

Implementation

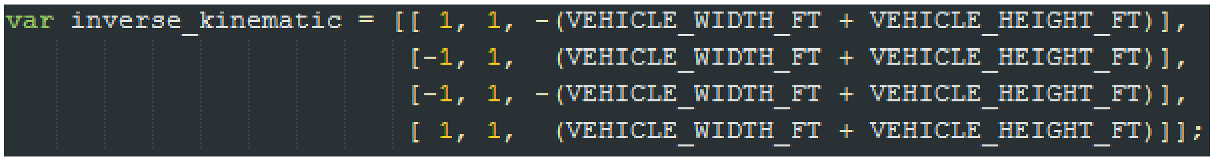


```
/* @brief Implementation of inverse kinematic equation
 */
function inverseKinematic(vx, vy, w)
{
   var arr = [vx, vy, w];
   var wheel_rot = [0, 0, 0, 0];

   for(var i = 0; i < arr.length; i++)
   {
      wheel_rot[0] += inverse_kinematic[0][i] * arr[i];
      wheel_rot[1] += inverse_kinematic[1][i] * arr[i];
      wheel_rot[2] += inverse_kinematic[2][i] * arr[i];
      wheel_rot[3] += inverse_kinematic[2][i] * arr[i];
   }

   for(var i = 0; i < wheel_rot.lenght; i++)
   {
      wheel_rot[i] *= (1/RADIUS);
   }
   return wheel_rot;
}
```

Code



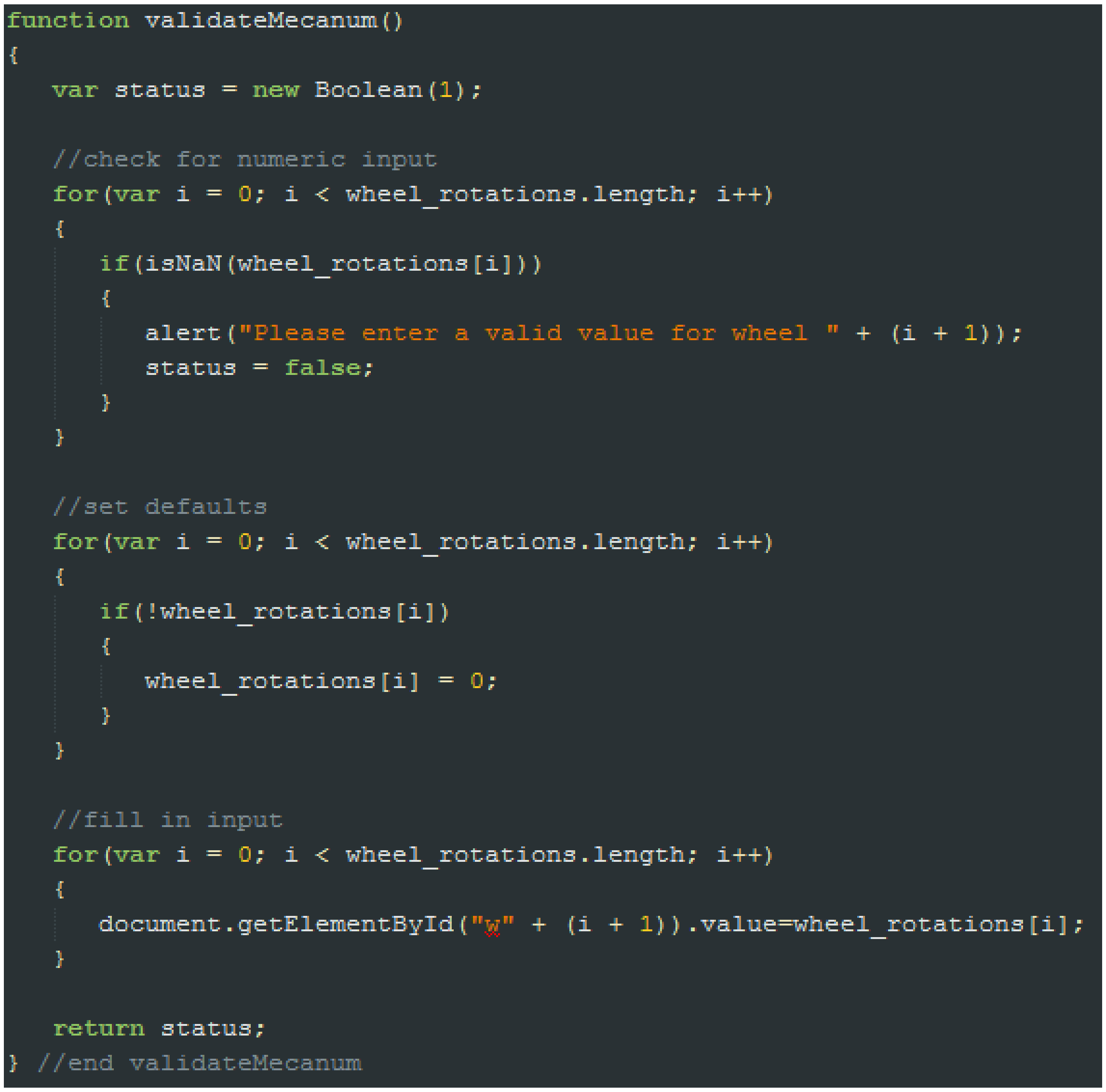
```
var inverse_kinematic = [[ 1, 1, -(VEHICLE_WIDTH_FT + VEHICLE_HEIGHT_FT)],
                         [-1, 1,  (VEHICLE_WIDTH_FT + VEHICLE_HEIGHT_FT)],
                         [-1, 1, -(VEHICLE_WIDTH_FT + VEHICLE_HEIGHT_FT)],
                         [ 1, 1,  (VEHICLE_WIDTH_FT + VEHICLE_HEIGHT_FT)]];
```

Control Algorithm

Calculating Animation Data

The following algorithm validates user input and calculates the data necessary to perform the animation of the vehicle. Animation data includes direction, rotation, and speed. The algorithm also considers the fact that the origin of the canvas is in the upper left corner of the screen whereas the global origin is initially in the center of the canvas.

1. Validate user input



```
function validateMecanum()
{
   var status = new Boolean(1);

   //check for numeric input
   for(var i = 0; i < wheel_rotations.length; i++)
   {
      if(isNaN(wheel_rotations[i]))
      {
         alert("Please enter a valid value for wheel " + (i + 1));
         status = false;
      }
   }

   //set defaults
   for(var i = 0; i < wheel_rotations.length; i++)
   {
      if(!wheel_rotations[i])
      {
         wheel_rotations[i] = 0;
      }
   }

   //fill in input
   for(var i = 0; i < wheel_rotations.length; i++)
   {
      document.getElementById("w" + (i + 1)).value=wheel_rotations[i];
   }

   return status;
} //end validateMecanum
```

2. Apply user input to forward kinematic equation

```
/* @brief Implementation of forward kinematic equation
 */
function forwardKinematic()
{
   //determine x and y component of velocity using forward kinematic equation
   //as well as rotation
   var matrix_mult_x = 0;
   var matrix_mult_y = 0;
   var matrix_mult_w = 0;
   for(var i = 0; i < wheel_rotations.length; i++)
   {
      matrix_mult_x += forward_kinematic[0][i] * wheel_rotations[i];
      matrix_mult_y += forward_kinematic[1][i] * wheel_rotations[i];
      matrix_mult_w += forward_kinematic[2][i] * wheel_rotations[i];
   }

   var Vx = (RADIUS/4) * matrix_mult_x;
   var Vy = (RADIUS/4) * matrix_mult_y;
   var Vw = (RADIUS/4) * matrix_mult_w;

  return [Vx, Vy, Vw];
}
```

3. Adjust rotation to align canvas coordinate system with Cartesian coordinate system

```
//rotation in degrees
ROTATION = -toDegrees(Vw);
```

4. Determine velocity vector from X and Y components using Pythagorean's Theorem

```
//determine velocity using Pythagoras' Theorem
SPEED = Math.sqrt(Math.pow(Vy,2) + Math.pow(Vx,2));
```

5. Check max vehicle speed

```
if(speedLimit() == false)
{
   return;
}
```

6. Draw path to be executed

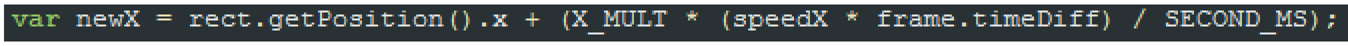
```
//draw path to be executed
drawPathToBeExec(DIRECTION, SPEED);
```

Animation

The following algorithm takes the animation data that was calculated above, adjusts it based on the frame rate, and passes it to the appropriate KineticJS function to perform the animation.

1. Determine new X coordinate based off of X component of velocity vector

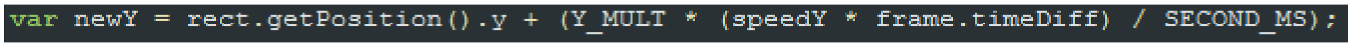
$$V_{C_X} = V_D \cos\left(\theta_D - \left(\theta_t + \Delta\theta\right)\right)$$



```
var newX = rect.getPosition().x + (X_MULT * (speedX * frame.timeDiff) / SECOND_MS);
```

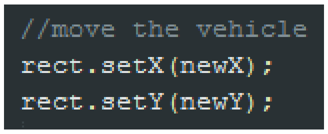
2. Determine new Y coordinate based off of Y component of velocity vector

$$V_{C_Y} = V_D \sin\left(\theta_D - \left(\theta_t + \Delta\theta\right)\right)$$



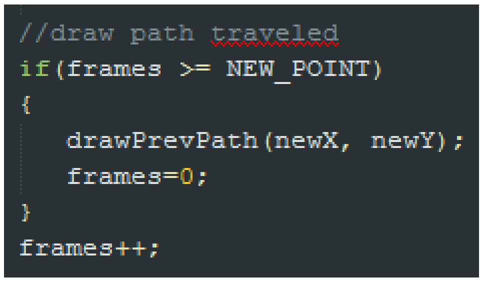
```
var newY = rect.getPosition().y + (Y_MULT * (speedY * frame.timeDiff) / SECOND_MS);
```

3. Move the vehicle



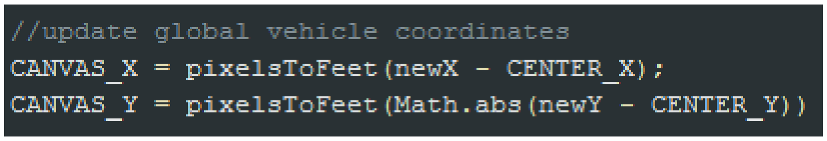
```
//move the vehicle
rect.setX(newX);
rect.setY(newY);
```

4. Draw the path the the vehicle has traversed



```
//draw path traveled
if(frames >= NEW_POINT)
{
   drawPrevPath(newX, newY);
   frames=0;
}
frames++;
```

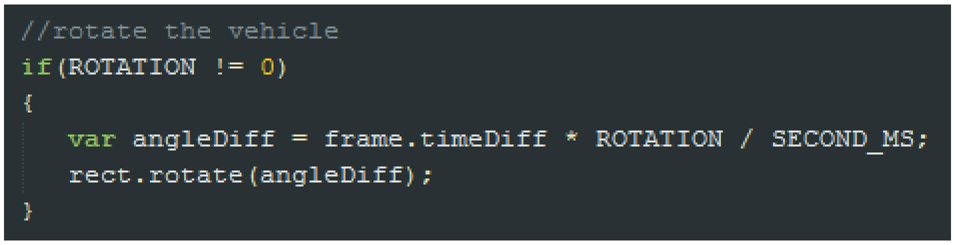
5. Update global coordinates



```
//update global vehicle coordinates
CANVAS_X = pixelsToFeet(newX - CENTER_X);
CANVAS_Y = pixelsToFeet(Math.abs(newY - CENTER_Y))
```

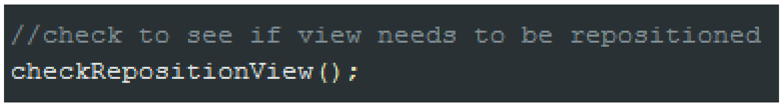
6. Rotate the vehicle

$$\omega_C = \frac{\Delta\theta}{t} \qquad \text{where } \Delta\theta = \theta_{t+1} - \theta_t$$



```
//rotate the vehicle
if(ROTATION != 0)
{
    var angleDiff = frame.timeDiff * ROTATION / SECOND_MS;
    rect.rotate(angleDiff);
}
```

7. Determine if the vehicle has come within 3 feet of canvas edge

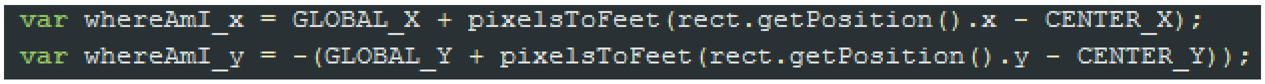


```
//check to see if view needs to be repositioned
checkRepositionView();
```

Waypoints

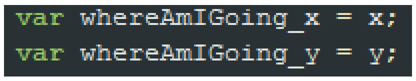
Waypoints leverage "The 3 Questions" of autonomous mobile robots: Where Am I?, Where Am I Going?, How Do I Get There?

1. Where Am I?



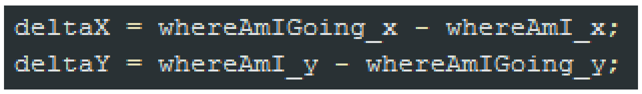
```
var whereAmI_x = GLOBAL_X + pixelsToFeet(rect.getPosition().x - CENTER_X);
var whereAmI_y = -(GLOBAL_Y + pixelsToFeet(rect.getPosition().y - CENTER_Y));
```

2. Where Am I Going?



```
var whereAmIGoing_x = x;
var whereAmIGoing_y = y;
```

3. How Do I Get There?



```
deltaX = whereAmIGoing_x - whereAmI_x;
deltaY = whereAmI_y - whereAmIGoing_y;
```

Circle

The control algorithm for the circle path mode is very similar to Mecanum mode with the following additions.

1. Calculate the circumference of the circle and determine the vehicle speed based on given time. This code is called only once prior to begining the animation.

```
//calculated distance and speed
var circumference = 2 * Math.PI * CIRCLE_RADIUS;
SPEED = circumference / TIME;
```

2. Determine how many degrees per frame the vehicle needs to move.

```
//set direction based on time
DIRECTION += (frame.timeDiff * (360/TIME)) / SECOND_MS;
```

3. Determine the new coordinates for the vehicle based on the X and Y components of the speed and the frame rate.

```
//determine x component of speed value
var speedX = feetToPixels(SPEED) * Math.cos(toRadians(DIRECTION));
var newX = rect.getPosition().x + (X_MULT * (speedX * frame.timeDiff) / SECOND_MS);

//determine y component of speed value
var speedY = feetToPixels(SPEED) * Math.sin(toRadians(DIRECTION));
var newY = rect.getPosition().y + (Y_MULT * (speedY * frame.timeDiff) / SECOND_MS);
```

Math Equations

- Pythagorean's Theorem

```
//determine velocity using Pythagoras' Theorem
SPEED = Math.sqrt(Math.pow(Vy,2) + Math.pow(Vx,2));
```