Application 1 - Runge Kutta Methods

José Mauricio Sevilla[1]
Andrés Camilo Sevilla[1]

[1]Universidad Nacional de Colombia

5 of March of 2016

Introduction

Methods

- Runge-Kutta Methods
- Remarks
- Generalization
- Runge-Kutta fourth Order
- Implementation
- Second Order Differential Equation

Introduction

Until now, we've been trying to understand how to give some instructions to the PC using C++, in this presentation, we're going to study this instructions applied to an specific problem, in this case it will be: solve differential equations.

Why solve differential equations?

On physics, we always are interested in *how things move*, it could be: in time, in position, or any other variable, the important part is that we always want to know how are the changes, and that's described by a differential equation.

Runge-Kutta Method

We can do it using a lot of methods, but, in this case, we choose a *Runge-Kutta* method.

To get there, we could make a step by step analysis, starting with the *Euler* methods, then getting a generalization with the *Runge-Kutta* methods, but, here we're interested in apply some concepts that aren't focused on the methods but getting results from any differential equation instead, that's why we'll jump there directly.

We have to keep in mind that to solving differential equation, it isn't enough having the equation, we need the *Initial Value Problem* associated, *i.e.*, *Initial conditions*[1], for example, let's consider a first order differential equation

$$y'(t) = f(t, y(t)) \tag{1}$$

where $y'(t)$ represents $\frac{dy}{dt}$ the derivative of y respect to t; And a initial condition:

$$y'(0) = f(0, y(0)) \tag{2}$$

This could be for any $t = t_0$, but generally $t_0 = 0$.

And now, we can wonder for the approximate solution of this problem, to do it, we consider the Taylor expansion of $y(t)$ around t.

[1] That could be boundary conditions in other context.

So, if we take this expansion until second order, we have:

$$y(t+h) = y(t) + hy'(t) + \frac{h^2}{2}y''(t) + \mathcal{O}(h^3) \tag{3}$$

And if we use our differential equation, we can use that $y'(t) = f(t, y(t))$, but, we also need the second derivative, we get it like:

$$\begin{aligned} y''(t) =& \partial_t f(t, y(y)) + \partial_x f(t, y(y))y'(t) \\ =& f_t(t, y(t)) + f_y(t, y(t))y'(t) \end{aligned}$$

So, we get an expression like

$$y''(t) = f_t(t, y(t)) + f_y(t, y(t))f(t, y(t)) \tag{4}$$

So, we get an expression for $y(t+h)$ like

$$y(t+h) = y(t) + hf(t,y(t)) + \frac{h^2}{2}\left[f_t(t,y(t)) + f_y(t,y(t))f(t,y(t))\right] + \mathcal{O}(h^3) \quad (5)$$

We can rewrite this last equation like:

$$y(t+h) = y(t) + \frac{h}{2}f(t,y(t)) + \frac{h}{2}\left[f(t,y(t)) + hf_t(t,y(t)) + hf_y(t,y(t))f(t,y(t))\right] + \mathcal{O}(h^3) \quad (6)$$

And here we can compare this expression with the 2-dimensional Taylor expansion of $f(t, y)$ as:

$$f(t + h_t, y + y_t) = f(t, y) + h_t \partial_t f(t, y) + h_y \partial_y f(t, y) + \mathcal{O}(h_t^2, h_y^2)$$

where h_t and h_y represent the infinitesimal change in t and y respectively

$$f(t + h_t, y + y_t) = f(t, y) + h_t f_t(t, y) + h_y f_y(t, y) + \mathcal{O}(h_t^2, h_y^2) \tag{7}$$

we can see that the term in the brackets is the same showed in the last equation but making $h_t = h$ and $h_y = hf(t, y(t))$

So, we have that

$$y(t+h) = y(t) + \frac{h}{2} f(t, y(t)) + \frac{h}{2} f(t+h, y + hf(t, y(t))) + \mathcal{O}(h^3) \quad (8)$$

or we can write it in a different notation making:

$$y_{n+1} \approx y_n + h\left(\frac{1}{2}k_1 + \frac{1}{2}k_2\right) \quad (9)$$

with

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + h, y_n + hf(t_n, y_n)) \\ &= f(t_n + h, y_n + hk_1) \end{aligned}$$

Remarks

We just took a first order differential equation, made a second Taylor expansion to find the solution in different times starting with the initial value problem, this is made in a iterative way to find the solution in the desired interval.

This approximation showed in (9), is called the second order *Runge-Kutta* method, due to a second order expansion is token, but it's also called the improved *Euler* Method, a first order expansion is called the *Euler* method, at third order is called *Euler* $3/8$ and fourth order is called *Runge-Kutta* fourth order.

Generalization

We can make a generalization of the *Runge-Kutta* methods, and we would get:

$$y_{n+1} = y_n + h\sum_{j=1}^{\mu} b_j k_j \tag{10}$$

with

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + hc_2, y_n + a_{21}hk_1) \\ &\vdots \\ k_\mu &= f(t_n + c_\mu h, y_n + h\sum_{j=1}^{\mu-1} a_{\mu j}k_j) \end{aligned}$$

And the constants a_i, b_i and c_i can be found, using the Butcher tableau.

Runge-Kutta fourth Order

Now, we are going to present one of the most used *Runge-Kutta* methods, the derivation is similar than showed to second order but making a fourth order expansion, now we have:

$$y_{n+1} = y_n + \frac{h}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right] \tag{11}$$

And

$$\begin{aligned} k_1 =& f(t_n, y_n) \\ k_2 =& f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \\ k_3 =& f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \\ k_4 =& f(t_n + h, y_n + hk_3) \end{aligned}$$

Implementation

We have now, an equation to find the evolution of the initial conditions of a first order differential equation, but, How do we write it in C++?, in this section, we'll show how to implement the expressions found before.
First, we need to think about the things we have to use

- A first-order differential equation.
- The initial value problem associated.
- A infinitesimal time step[2] h.
- The number of iterations, that combined with h gives the interval where the solution is found.
- Find the values of k_1, k_2, k_3 and k_4 at every time-step to get the evolution of t and y.

[2] Like we did a Taylor expansion using this parameter, this have to be small enough compared with the changes of the function.

We are going to construct a first code:

```
#include<iostream>
#include<math.h>
double f(double t, double y){
return cos(t);}
int main(){
double h = 0.1; double t=0;     double y=0;
double k1,k2,k3,k4;
for (int i=0;i<=200;i++){
        k1=f(t,y);
        k2=f(t+h/2,y+h*k1/2);
        k3=f(t+h/2,y+h*k2/2);
        k4=f(t+h,y+h*k3);
        y=y+(k1+2*k2+2*k3+k4)*h/6;
        t=t+h;
        std::cout<<t<<'\t'<<y<<std::endl;}
return 0;}
```

Now we're going to explain this preliminary code, first we have the libraries definition

```
#include<iostream>
#include<math.h>
```

We're using `iostream` to print in the terminal the values (in this case is in the line `std::cout<<t<< '\t'<<y<<std::endl;`), and `math.h` to use the cosine function. Then, we have the function definition

```
double f(double t, double y){
return cos(t);}
```

we choose the cosine function, but we could use any other, and in principle, it could depend of t and y.

Then we have the main function the definition of k_1, k_2, k_3 and k_4, , and one of the most important things, the initial conditions.

```
int main(){
double h = 0.1; double t=0; double y=0;
double k1,k2,k3,k4;
```

The then we have the implementation of the method, the loop to find 200 time-steps, and the finalization of the program

```
for (int i=0;i<=200;i++){
        k1=f(t,y);
        k2=f(t+h/2,y+h*k1/2);
        k3=f(t+h/2,y+h*k2/2);
        k4=f(t+h,y+h*k3);
        y=y+(k1+2*k2+2*k3+k4)*h/6;
        t=t+h;
        std::cout<<t<<'\t'<<y<<std::endl;}
return 0;}
```

Now, let's use our code, first, we need to save it, in this example we're going to save the file as `example1.cpp`, in the following folder

```
~/Documents/examples/
```

to get there, we use the command `cd`

```
cd ~/Documents/examples/
```

then, we have to compile the example

```
g++ example1.cpp
```

or if we want a different name than `a.out` for the executable file like `example1`

```
g++ example1.cpp -o example1
```

and then we run the example:

```
./example1
```

but, doing that we just get a data on the shell, let's do a plot with it so we can interpret the results, to do it, we need to save this data, let's save them in `data1.dat`

```
./example1 > data1.dat
```

and to plot it, we're going to use `Gnuplot`, so, first we open it by typing

```
gnuplot
```

Once gnuplot is opened, we just give the instructions we want to make our plot

```
set xlabel "t"
set ylabel "y(t)"
set title  "Results"
set terminal pdf
set output "Plot1.pdf"
plot "data1.dat" title "data"
set terminal qt
replot
```

The plot with the results is:

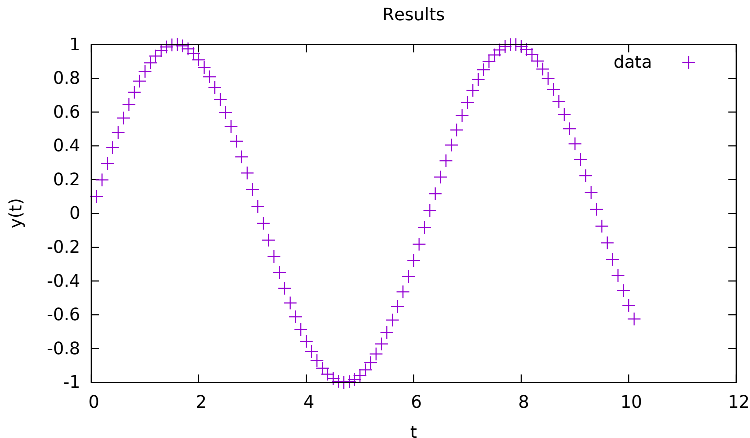

Figure 1: Graphic saved in `Plot.pdf`.

But, we can make different the program, for example, if we have to solve many *Differential Equations* in our program, doing it like we just showed, we'd had to write the method as many times as needed, but,

What if we use a function to that?

```
#include<iostream>
#include<math.h>
void Rk(double &,double &,double &,double );
double f(double ,double );
int main(int argc, char* argv[]){
double xx0,xxf,yy0,hh;
int n;
xx0=0; xxf=10; yy0=0; n=100;
hh=(xxf-xx0)/n;
for (int i=0; i<=n; i++){
        Rk(xx0,xxf,yy0,hh);
        std::cout<<xx0<<'\t'<<yy0<<std::endl;}
return 0;}
double f(double x,double y){
        return cos(x);}
void Rk(float &x0,double &xf,double &y0,double h){
        double k1,k2,k3,k4;
        k1=f(x0,y0);    k2=f(x0+h/2,y0+h*k1/2);
        k3=f(x0+h/2,y0+h*k2/2); k4=f(x0+h,y0+h*k3);
        y0=y0+(k1+2*k2+2*k3+k4)*h/6;    x0=x0+h;}
```

Now, let's explain the code, first, we have the libraries declaration

```
#include<iostream>
#include<math.h>
```

After that, we declare the functions we are going to use, here we must note that the function `Rk` (*Runge Kutta*) is `void` but their parameters are passed by reference, we are going to explain it later.

```
void Rk(double &,double &,double &,double );
double f(double ,double );
```

Now, we make the main function, and, until now, we never pass parameters, but in here we use two and one of them is an array pointer, we are going to use them later, but let's get familiar with it.

```
int main(int argc, char* argv[]){
double xx0,xxf,yy0,hh;
int n;
xx0=0; xxf=10; yy0=0; n=100;
hh=(xxf-xx0)/n;
for (int i=0; i<=n; i++){
 Rk(xx0,xxf,yy0,hh);
 std::cout<<xx0<<'\t'<<yy0<<std::endl;}
return 0;}
```

Here we have some double variables, those are the interval where we want the solution [xx0,xxf] and the initial condition yy0.

Then, we have the fourth order *Runge Kutta* method implementation, where, like we are using a `void` function, we have no `return`, but how do we get the solution if we are no returning the new values at every step?

```
void Rk(double &x0,double &xf,double &y0,double h){
        double k1,k2,k3,k4;
        k1=f(x0,y0);
        k2=f(x0+h/2,y0+h*k1/2);
        k3=f(x0+h/2,y0+h*k2/2);
        k4=f(x0+h,y0+h*k3);
        y0=y0+(k1+2*k2+2*k3+k4)*h/6;
        x0=x0+h;}
```

that's why we are using values passed by reference. We can test this program in the same way that we did in the last case, let's make the plot.

Second Order *Differential Equation*

Now, let's consider a higher order differential equation, for example, a well known one with a grate physical meaning, the *Harmonic Oscillator*, and if we consider no friction and no external force:

$$\frac{d^2x}{dt^2} + \omega_0^2 x = 0 \tag{12}$$

We can rewrite this equation as a system of first order differential equations like:

$$\frac{dv}{dt} = -\omega_0^2 x \tag{13}$$

$$\frac{dx}{dt} = v \tag{14}$$

And we must use a *Runge-Kutta* method for each one, let's use the fourth order *Runge-Kutta* method.

Like this is a second order differential equation, we need two initial conditions to solve the problem, $x(0)$ and $v(0)$.

We have to keep in mind that due to this two equations are coupled, we must solve them at the same time, so, in the *Runge-Kutta* method, we must introduce other constants like k_i, but we are going to call them ℓ_i, let's consider

$$\frac{dv}{dt} = f(x, v, t) \tag{15}$$

$$\frac{dx}{dt} = g(x, v, t) \tag{16}$$

like we did in the previous case. Let's see the code using $x(0) = 1{,}0$ and $v(0) = 0{,}0$:

Code part I

```
#include<iostream>
#include<math.h>
void time_step(double &,double &,double &,double );
double f(double ,double );
double g(double );
int main(int argc, char* argv[]){
double t0,tf,xx,vv,hh,gg;
int n;
t0=0.0;
tf=100.0;
xx=1.0;
vv=0.0;
n=500;
hh=(tf-t0)/n;
for (int i=0; i<=n; i++){
        time_step(t0,xx,vv,hh);
        std::cout<<t0<<'\t'<<xx<<'\t'<<vv<<std::endl;}
return 0;
}
```

Code part II

```
double f(double x,double v,double t){
        double w0=1.0;
        return -w0*x;}
double g(double x,double v,double t){
        return v;}
void time_step(double & t,double &x,double &v,double h){
        double k1,k2,k3,k4;
        double l1,l2,l3,l4;
        k1=h*g(x, v, t);
        l1=h*f(x, v, t);
        k2=h*g(x+k1/2.0, v+l1/2.0, t+h/2.0);
        l2=h*f(x+k1/2.0, v+l1/2.0, t+h/2.0);
        k3=h*g(x+k2/2.0, v+l2/2.0, t+h/2.0);
        l3=h*f(x+k2/2.0, v+l2/2.0, t+h/2.0);
        k4=h*g(x+k3, v+l3, t+h);
        l4=h*f(x+k3, v+l3, t+h);
        x+=(k1+2.0*k2+2.0*k3+k4)/6.0;
        v+=(l1+2.0*l2+2.0*l3+l4)/6.0;
        t+=h;    }
```

Let's test the code by using the analytical solution:

$$x(t) = A\cos(\omega t) + B\sin(\omega t) \tag{17}$$

$x(0) = 1{,}0$, $\omega = 1$, we'd have:

$$x(t) = A\cos(0) + B\sin(0) = A = 1{,}0 \tag{18}$$

y como $v(0) = 0{,}0$

$$\dot{x}(0) = -\sin(0) + B\cos(0) = B = 0{,}0 \tag{19}$$

Por lo tanto, la solución es:

$$x(t) = \cos(t) \tag{20}$$

We get as result:

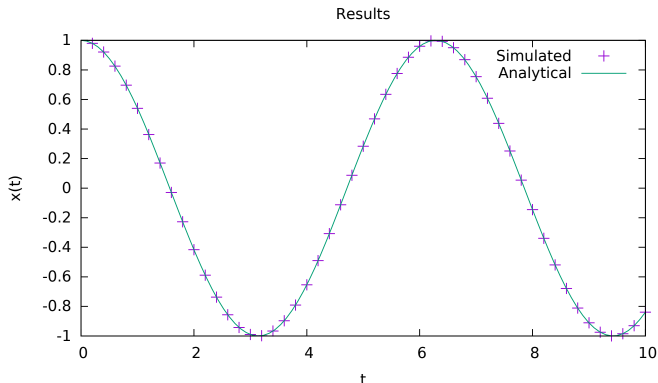

Figure 2: Plot of the results, simulated and analytical

we also can make the plot of the velocity in function of time

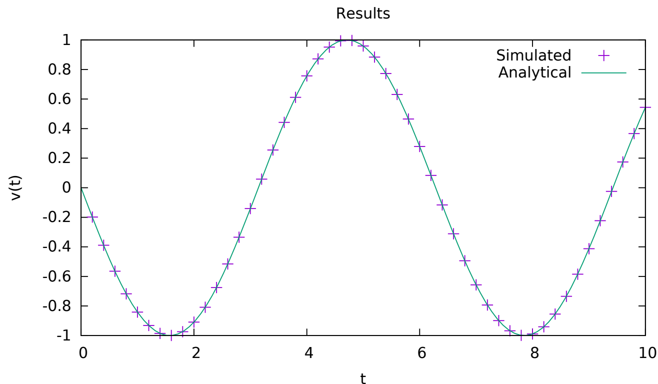

Figure 3: Velocity in function of time, simulated and analytical

And the last result we can check the phase-space:

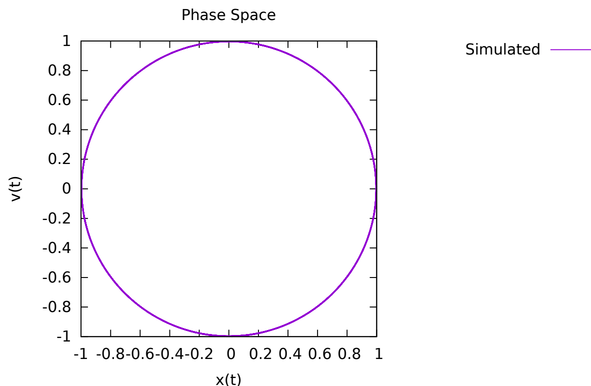

Figure 4: Velocity in function of time, simulated and analytical

the next thing we may wonder is, using this program, how do we simulate an *Dumped Harmonic Oscillator*? We just have to write different the functions $f(x, v)$ and $g(x, v)$

$$\frac{dv}{dt} = -\omega_0^2 x - \gamma v \tag{21}$$

$$\frac{dx}{dt} = v \tag{22}$$

Let's use $\gamma = 0{,}15$[3] and run our simulation, then, let's check the results using the analytical solution.

[3] There exist a reason to take this value of γ and it is that we want the underdamped case.

For the position $x(t)$

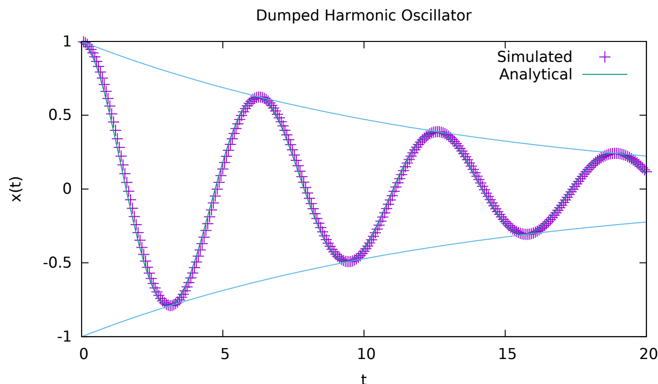

Figure 5: Position in function of time, simulated and analytical for Dumping Harmonic Oscillator

For the Velocity $v(t)$

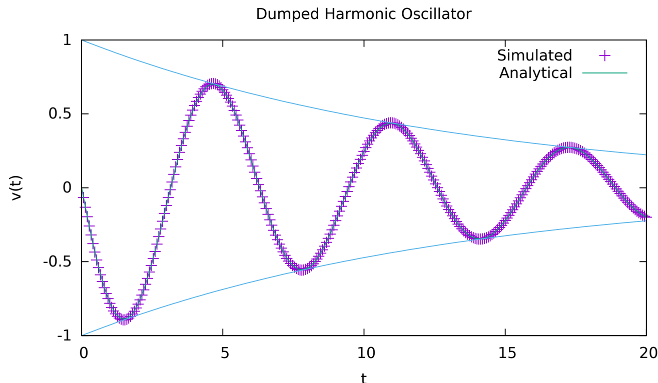

Figure 6: Velocity in function of time, simulated and analytical for Dumping Harmonic Oscillator

For the Phase-Space

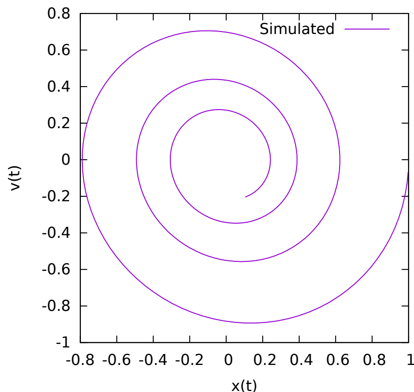

Figure 7: Phase-Space for Dumping Harmonic Oscillator

Now, we can study one last case, that is add an external force, let's use an harmonic external force:

$$\frac{dv}{dt} = -\omega_0^2 x - \gamma v - F\cos(\Omega t) \quad (23)$$

$$\frac{dx}{dt} = v \quad (24)$$

Let's see the results

For the position $x(t)$

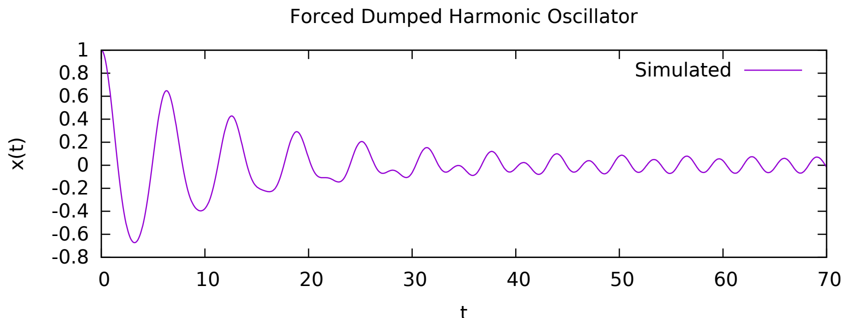

Figure 8: Position in function of time, simulated and analytical for Forced Dumping Harmonic Oscillator

For the Velocity $v(t)$

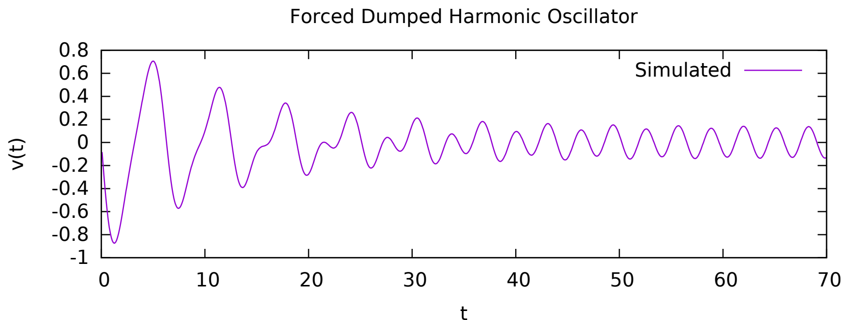

Figure 9: Velocity in function of time, simulated and analytical for Forced Dumping Harmonic Oscillator

For the Phase-Space

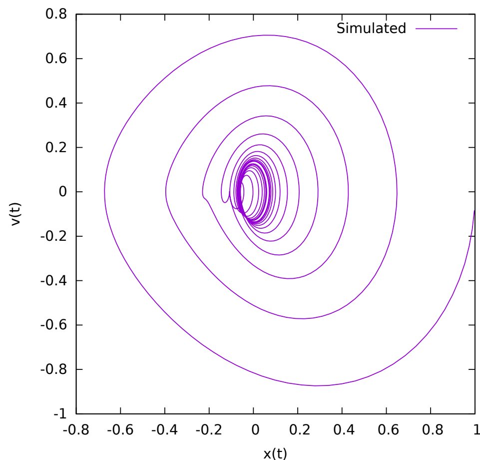

Figure 10: Phase-Space for Forced Dumping Harmonic Oscillator