## Описание метода решения задачи Коши

Для решения задачи Коши от двух переменных выбран метод Эйлера. Он заключается в том, что мы строим точки $x_n, y_n, t_n$ по следующему индективному правилу:

- $x_0, y_0$ – начальная точка из условия задачи Коши, $t_0 = 0$,
- $x_{n+1} = x_n + t * f_x(x_n, y_n, t_n), \quad y_{n+1} = y_n + t * f_y(x_n, y_n, t_n), \quad t_{n+1} = t_n + t,$

где $t$ – значение временного шага. Чем оно меньше, тем точнее решение, полученное в результате применения метода.

Полученные точки $(x_n, y_n)$ образуют табулированное численное решение.

## Единственность решения

Точное решение задачи Коши единственно по теореме о единственности.

Наше численное решение для заданного заранее параметра $t$ также обладает свойством единственности, так как если бы фазовые кривые пересекались, возникло бы противоречие с тем, что для точки пересечения алгоритм построения следующей точки решения детерминирован.

## Применение решения для конкретной задачи Коши

Рассмотрим задачу $x' = y, \quad y' = -x, \quad x_0 = 1, \quad y_0 = 0.$

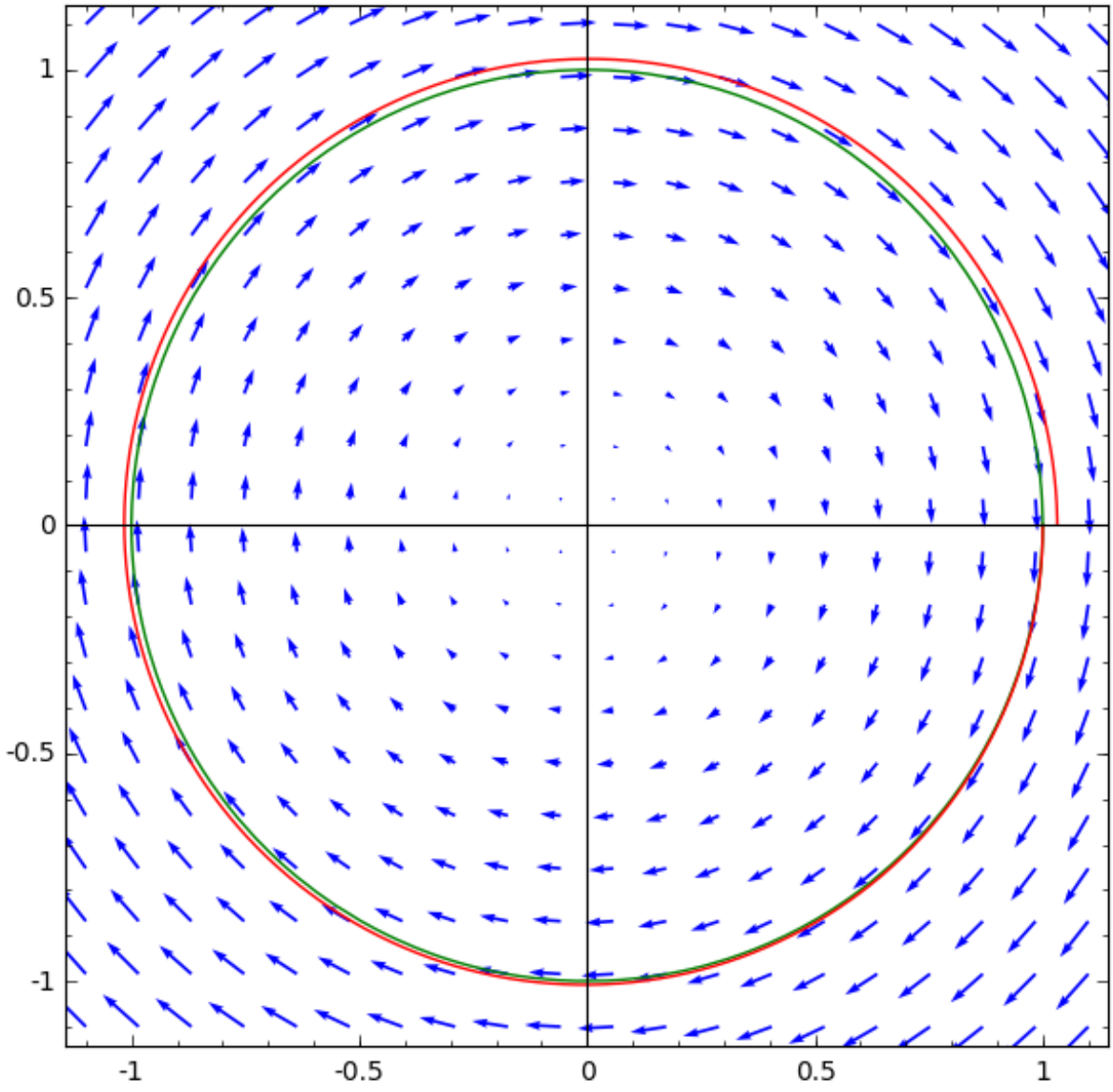
Ее точное решение: $x = cos(t), \quad y = -sin(t)$. Без заданного начального условия, фазовые кривые имеют вид $x = C\cos(t), \quad y = -C\sin(t)$, являются окружностями всех радиусов вокруг начала координат.

Код, строящий графики

```
class DiffEq(object):
   def __init__(self, fx, fy, start_t, end_t, step_t, x0, y0):
      self.fx = fx
      self.fy = fy
      self.start_t = start_t
      self.end_t = end_t
      self.step_t = step_t
      self.x0 = x0
      self.y0 = y0

   def build_solution(self):
      x = []
      y = []
      x_y_data = []
      cur_t = self.start_t
      prev_x = self.x0
      prev_y = self.y0
      prev_t = cur_t
      while cur_t < self.end_t:
```

```
        cur_x = prev_x + (cur_t - prev_t) * self.fx(x = prev_x, y = prev_y, t =
            prev_t)
        cur_y = prev_y + (cur_t - prev_t) * self.fy(x = prev_x, y = prev_y, t =
            prev_t)
        x.append((cur_t, cur_x))
        y.append((cur_t, cur_y))
        x_y_data.append((cur_x, cur_y))
        prev_x = cur_x
        prev_y = cur_y
        prev_t = cur_t

        cur_t += self.step_t
      self.x = x
      self.y = y
      self.x_y_data = x_y_data


x = var('x')
y = var('y')
t = var('t')
g = Graphics()
g += plot_vector_field((y, -x), (x,-1.1,1.1), (y,-1.1,1.1), color="blue")
g += parametric_plot((cos(t),sin(t)),(t, 0, 2*pi),color="green")
diff_eq = DiffEq(y, -x, 0, 2*pi, 0.01, 1, 0)
diff_eq.build_solution()
g += list_plot(diff_eq.x_y_data, plotjoined=True,color="red")
g.save("diff_eq_solution.png")
```

Синим цветом обозначено веректорное поле, зеленым – идеальное аналитическое решение, красным – приближенное решение. Видно, что оно отличается от идеального решения.




## Оценка погрешности метода

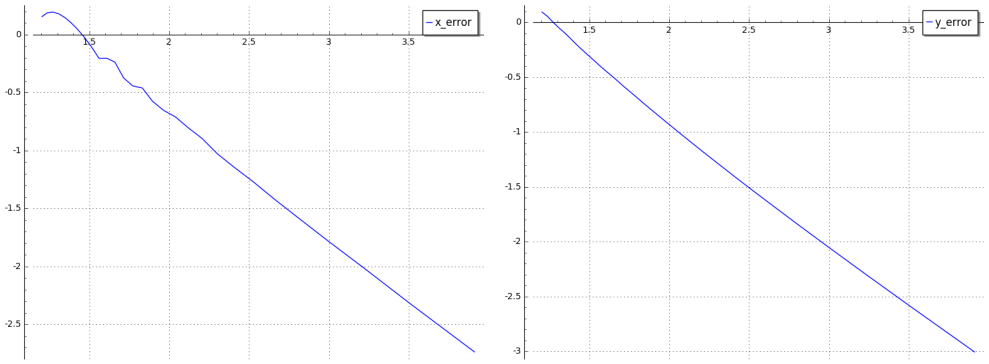
Для выше рассмотренной задачи построим графики логарифма ошибки по норме Чебышева от логарифма шага алгоритма.

Код, который строит график ошибки

```
x = var('x')
```

```
y = var('y')
t = var('t')
h = 0.3
log_error_points = {}
while h > 0.01:
   diff_eq = DiffEq(y, -x, 0, 2*pi, h, 1, 0)
   diff_eq.build_solution()

   print "for h={h}".format(h=h)
   for (real_func, approx_func, var_name) in ((cos(t), diff_eq.x, 'x'),
       (-sin(t), diff_eq.y, 'y')):
      max_norm = 0
      cur_t = 0
      for (t_, x_) in approx_func:
         if max_norm < abs(real_func(t_) - x_):
            max_norm = abs(real_func(t_) - x_)

      log_error_points.setdefault(var_name, []).append((-log(h), log(max_norm)))

   h -= 0.01

for var_name in ('x', 'y'):
   g = Graphics()
   g += list_plot(log_error_points[var_name], plotjoined=True,
       legend_label="{var_name}_error".format(var_name=var_name), gridlines=True)
   g.save("diff_eq_{var_name}_error.png".format(var_name=var_name))
```

Графики ошибок:




Из них незатруднительно видеть, что метод имеет первый порядок ошибки.