

A General ODE & PDE Solver Using Picard's Method

James H. Money

Undergraduate Student

James Madison University

Harrisonburg, VA 22807

jmoney@math.jmu.edu

ABSTRACT

Initial value ODE's and PDE's can be converted to an integral equation and this integral equation can be solved using Picard's method of successive approximations. Generally the integration becomes impossible. I will show how to convert the ODE's and PDE's to polynomial form. Using this polynomial form the integrations can be calculated symbolically. I will present a code that can generate the Picard iterates to any general initial value ODE or PDE.

Picard's Method for solving ODEs and PDEs

Recent discoveries by Dr. Edgar Parker and Dr. James Sochacki at James Madison University has led to a new way of solving ordinary differential equations(ODEs). I will present the basic approach here that is described in their papers.[REF]

Consider the initial value ODE

$$y'(t) = F(t, y(t)); y(t_0) = y_0,$$

where $F : \mathfrak{R}^{n+1} \rightarrow \mathfrak{R}^n$, and its equivalent integral equation:

$$y(t) = y_0 + \int_{t_0}^t F(s, y(s))ds$$

Picard's method iterates on the integral equation, by defining:

$$\begin{aligned} y_1(t) &= y_0 \\ y_{k+1}(t) &= y_0 + \int_{t_0}^t (F(s, y_k(s)))ds \end{aligned}$$

To use the theorems of Sochacki and Parker, F has to be a polynomial and t_0 has to be 0. For an example:

$$\begin{aligned} y' &= \cos(y) + \sin(t) \\ y(0) &= 0 \end{aligned}$$

Now, on the computer we will want to use only polynomial based equations, so we transform the $\cos(y)$ and $\sin(t)$. For this, we will employ an old practice of calculus, namely substitutions. We are now going to make a system of equations by letting:

$$\begin{aligned} U &= \cos(y) \\ V &= \sin(t) \end{aligned}$$

Making these substitutions and computing the derivatives gives the following system of equations:

$$\begin{aligned} y' &= U + V \\ U' &= -\sin(y)y' \\ V' &= \cos(t) \end{aligned}$$

We're not quite done yet, we need to substitute for $-\sin(y)y'$ and $\cos(t)$:

$$W = \sin(y)$$

$$Z = \cos(t)$$

So our system is now:

$$\begin{aligned} y' &= U + V \\ U' &= -\sin(y)y' \rightarrow U' = -W * (U + V) \\ V' &= \cos(t) \rightarrow V' = Z \\ W' &= \cos(y)y' \rightarrow W' = U * (U + V) \\ Z' &= -\sin(t) \rightarrow Z' = -V \end{aligned}$$

We can now solve this system of equations using Picard's Method. Sochacki and Parker's method is valid only for initial conditions at 0. How do we achieve this?

First, we substitute $s = t - t_0$ where t_0 is the initial condition time. We then define a new set of equations, u_1 and u_2 such that:

$$\begin{aligned} u_1 &= y(s + t_0) \\ u_2 &= s + t_0 \end{aligned}$$

Now, we can take the derivative for each of these, and get the new initial conditions by plugging in $t = t_0$ in $s = t - t_0$:

$$\begin{aligned} u'_1 &= y'(s + t_0) \\ u'_2 &= 1 \end{aligned}$$

Now we can make further substitutions if the u'_1 contains any non-polynomial components.

For example, lets take the above equation and compensate for $y(1) = 1$:

$$\begin{aligned} y' &= \cos(y) + \sin(t) \\ y(1) &= 1 \end{aligned}$$

We convert to u_1 and u_2 :

$$\begin{aligned} u_1 &= y(s + 1) \\ u_2 &= s + 1 \end{aligned}$$

$$\begin{aligned} u'_1 &= y'(s + 1) = \cos(y(s + 1)) + \sin(s + 1) = \cos(u_1) + \sin(u_2) \\ u'_2 &= 1 \end{aligned}$$

And then we do our substitutions as before:

$$u_3 = \cos(u_1)$$

$$u_4 = \sin(u_2)$$

$$u'_1 = u_3 + u_4$$

$$u'_2 = 1$$

$$u'_3 = -\sin(u_1)u'_1 = -\sin(u_1)(u_3 + u_4)$$

$$u'_4 = \cos(u_2)u'_2 = \cos(u_2)$$

And we go again:

$$u_5 = \sin(u_1)$$

$$u_6 = \cos(u_2)$$

$$u'_1 = u_3 + u_4$$

$$u'_2 = 1$$

$$u'_3 = -u_5(u_3 + u_4)$$

$$u'_4 = u_6$$

$$u'_5 = \cos(u_1)u'_1 = u_3(u_3 + u_4)$$

$$u'_6 = -\sin(u_2)u'_2 = -u_4$$

With the initial conditions:

$$u_1(0) = 1$$

$$u_2(0) = 1$$

$$u_3(0) = \cos(1)$$

$$u_4(0) = \sin(1)$$

$$u_5(0) = \sin(1)$$

$$u_6(0) = \cos(1)$$

How the program is Implemented

The program that I produced for solving ODEs and PDEs using Picard iteration is written in C++ and includes parallelizations using PVM 3.3.11. The program is divided into several sections:

- Mathematical language parser
- Mathematical integrater and substituter
- Parallel modifications for use with PVM

Language Parser

The language parser for the solver is a standard implementation of a recursive descent parser. The language provides you with the flexibility to enter just about any type of equation you would require for Picard iteration. Here is the BNF table for the language:

```
<S> → sp[<number>];var[<number>];<equationset> <equationset2><EOF>
<equationset> → <equation> <init>
<equationset2> → <equation> <init>
<equationset2> → e
<equation> → var[<num>] :=<complexterm>;
<complexterm> → <factor> + <factor>
<complexterm> → <factor>
<factor> → <term> * <term>
<factor> → <term>
<term> → <symbol> <term>
<term> → <symbol>
<symbol> → var[<num>]<power>
<symbol> → <float> var[<num>]<power>
<symbol> → sp[<num>]<power>
<symbol> → <float> sp[<num>]<power>
>symbol> → D(<complexterm>,sp[<num>])
<symbol> → (<complexterm>)
<power> → <num>
<power> → e
<num> → 0..9<num2>
<num2> → 0..9 <num2>
<num2> → e
```

$\langle \text{float} \rangle \rightarrow \langle \text{num} \rangle \langle \text{exp} \rangle$
 $\langle \text{float} \rangle \rightarrow \langle \text{num} \rangle . \text{num} \langle \text{exp} \rangle$
 $\langle \text{float} \rangle \rightarrow - \langle \text{num} \rangle \langle \text{exp} \rangle$
 $\langle \text{float} \rangle \rightarrow - \langle \text{num} \rangle . \text{num} \langle \text{exp} \rangle$
 $\langle \text{exp} \rangle \rightarrow e \langle \text{sign} \rangle \langle \text{num} \rangle$
 $\langle \text{exp} \rangle \rightarrow e$
 $\langle \text{sign} \rangle \rightarrow +$
 $\langle \text{sign} \rangle \rightarrow -$
 $\langle \text{init} \rangle \rightarrow \text{var}[\langle \text{num} \rangle] () := \langle \text{simpleterm} \rangle ;$
 $\langle \text{simpleterm} \rangle \rightarrow \langle \text{float} \rangle \langle \text{simplesym} \rangle$
 $\langle \text{simpleterm} \rangle \rightarrow \langle \text{simplesym} \rangle$
 $\langle \text{simplesym} \rangle \rightarrow \text{sp}[\langle \text{num} \rangle] \langle \text{power} \rangle \langle \text{simplesym} \rangle$
 $\langle \text{simplesym} \rangle \rightarrow e$

Given this language specification we can generate the input for the first example:

$$\begin{aligned}
 y' &= U + V \\
 U' &= -W * (U + V) \\
 V' &= Z \\
 W' &= U * (U + V) \\
 Z' &= -V
 \end{aligned}$$

The resulting input would be:

$$\begin{aligned}
 &sp[0]; \\
 &var[5]; \\
 &var[1] := var[2] + var[3]; \\
 &var[1]() := 0; \\
 &var[2] := -1.0var[4] * var[2] + -1.0var[4] * var[3]; \\
 &var[2]() := 1; \\
 &var[3] := var[5]; \\
 &var[3]() := 0; \\
 &var[4] := var[2] * (var[2] + var[3]); \\
 &var[4]() := 0;
 \end{aligned}$$

$$\text{var}[5] := -1.0\text{var}[3];$$

$$\text{var}[5]() := 1;$$

As an example, the number of Picard iterates I used is 5 and the time step is 0.05. To iterate on the time step I use $\bar{y}(0.05)$ as the new initial condition ($\bar{y} = (y, U, V, W, Z)$).

Here is the plot of the resulting answer using Maple:

Substituter and Integrater

After all the information for the equations is read into memory and set up for processing, the next step involves substituting and integrating for the actual Picard process. The program begins for the first equation by plugging in the correct values for $\text{var}[1]$, $\text{var}[2]$, etc., After it does that, it integrates the resulting simple equation that contains no $\text{var}[1]$, $\text{var}[2]$, etc. references. Finally, it adds the initial value back to the answer. The program then repeats it for the next equation. For example, let's take the first example as our program input:

The first equation is :

$$\text{var}[1] := 1\text{var}[2] + 1\text{var}[3];$$

And the initial values are:

$$\text{var}[2]() := 1;$$

$$var[3]() := 0;$$

So the result after plugging in is:

$$1$$

And after we integrate we get:

$$t$$

Then we add on the initial value for $var[1]() := 0$ and get:

$$t$$

We can now do this several times, updating the plugged in value, but not changing the initial condition. We can also update the initial condition for the new h time step value by plugging h into t:

$$y_{k+1}(t) = y_0 + \int_0^t (F(s, y_k(s))) ds$$

Then set y_0 to:

$$y_0 = y_k(h)$$

PDEs

For PDEs, we modify our process slightly. Consider:

$$U_{tt} = \frac{\partial}{\partial x}(x^2 U_x) + \sin(U)$$

$$U(0) = \sin(x)$$

$$U_t(0) = 0$$

First, we make a substitution so that:

$$V = U_t$$

And, then we take the derivative:

$$V_t = U_{tt} = \frac{\partial}{\partial x}(x^2 U_x) + \sin(U)$$

Now, we proceed as before making substitutions:

$$V_t = U_{tt} = \frac{\partial}{\partial x}(x^2 U_x) + W$$

$$W = \sin(U)$$

$$Z = \cos(U)$$

and the derivatives are:

$$U_t = V$$

$$V_t = \frac{\partial}{\partial x}(x^2 U_x) + W$$

$$W_t = \cos(U)U_t = ZV$$

$$Z_t = -\sin(U)U_t = -WV$$

And the initial conditions are:

$$U(0) = \sin(x)$$

$$V(0) = 0$$

$$W(0) = \sin(\sin(x))$$

$$Z(0) = \cos(\sin(x))$$

In order to use my code the initial conditions have to be approximated by Maclaurin polynomials. This will be seen in the examples.

Truncation methods

In order to save memory on the computer and to increase the speed of processing, we want to only perform calculations that have "significant" impact on the numerical solutions. We do this by considering the coefficients of the space variables and the time variables.

It is simple to do this for the space variables. We introduce a minimal coefficient we feel that is sufficient to weed out the numbers that would evaluate to significantly small value.

To truncate the time variable equations during integration, we only keep the terms that are below the power $n+1$, where n is the n th iteration of the Picard process. Any number higher than that is not necessary for computational purposes, since the Maclaurin polynomial in time is made up of the powers less than or equal to n .

Parallel Processing

To parallel process Picard iterations, it is fairly simple. Since the initial data does not change during the iteration, but only after all the equations have been iterated, you can distribute the equations "evenly" among the machines.

To accomplish this task, I used PVM v3.3.11 on a set of SGI IRIX 6.2 boxes with 64MB of RAM. I passed a single equation, the initial value for the equation, and values to plug in for each variable to a child process to handle on a remote machine. This child process under PVM, plugs in the data, integrates, and adds the initial condition to the answer. The result

is returned to the controlling process. The controlling process in turn updates, all the values for the second iteration, and repeats the step.

One current problems lies in the parallel implementation. The reason is the algorithm cannot predict which machines will finish first, and thus, a machine might wait a significant time period for another machine to finish. It might be possible in future versions to distribute the load even more by dividing up the plugging in and integrating operations among several machines.

Examples

1.) Lorenz attractor. The equation for the Lorenz attractor is:

$$X' = \sigma(Y - X)$$

$$Y' = -XZ + r_a X - Y$$

$$Z' = XY + bZ$$

I use:

$$\sigma = 10$$

$$r_a = 28$$

$$b = \frac{8}{3}$$

$$X(0) = 1$$

$$Y(0) = 0$$

$$Z(0) = 0$$

The input file is:

`sp[0];`

`var[3];`

`var[1] := 10var[2] + -10var[1];`

`var[1]() := 1;`

`var[2] := -1var[1]var[3] + 28var[1] + -1var[2];`

`var[2]() := 0;`

`var[3] := var[1]var[2] + -2.6666666666667var[3];`

`var[3]() := 0;`

The corresponding graphed output is:

2.) Rossler Equation. The equation is:

$$X' = -Y - Z$$

$$Y' = X + pY$$

$$Z' = q + XZ - rZ$$

I use:

$$p = q = 0.2$$

$$r = -0.5$$

$$X(0) = 0$$

$$Y(0) = 0$$

$$Z(0) = 0$$

The input file is:

$$sp[0];$$

$$\begin{aligned}
& var[3]; \\
var[1] & := -1.0var[2] + -1.0var[3]; \\
var[1]() & := 0; \\
var[2] & := var[1] + 0.2var[2]; \\
var[2]() & := 0; \\
var[3] & := 0.2 + var[1] * var[3] + -0.5var[3]; \\
var[3]() & := 0;
\end{aligned}$$

The resulting plot is:

3.) One Way Wave equation:

$$U_t = U_x$$

The initial value is:

$$U(0) = \sin(x)$$

The input file is:

$$sp[1];$$

$$\begin{aligned} & var[1]; \\ & var[1] := D(var[1], sp[1]); \\ & var[1]() := 1.0sp[1] + -0.1666666667sp[1]^3 + 0.8333333333e - 2sp[1]^5 + \\ & \quad \text{..up to 80 terms} \end{aligned}$$

The before picture is:

The after picture is:

4.)Wave Equation:

$$U_t = V$$
$$V_t = \frac{\partial}{\partial x} U_x$$

With initial values:

$$U(0) = e^{-x^2}$$
$$V(0) = 0$$

The input file is :

$$sp[1];$$
$$var[2];$$
$$var[1] := var[2];$$
$$var[1]() := 1 + -1sp[1]^2 + 0.5sp[1]^4 + \dots \text{up to 80 terms}$$
$$var[2] := D(D(var[1], sp[1]), sp[1]);$$
$$var[2]() := 0;$$

The plot before is:

The plot after is:

5.) Wave Equation:

$$U_t = V$$
$$V_t = \frac{\partial}{\partial x}(x^2 * U_x)$$

With the initial values:

$$U(0) = \sin(x)$$
$$V(0) = 0$$

We use the input file:

```
sp[1];  
var[2];  
var[1] := var[2];  
var[1]() := 1.0sp[1] + -0.16666666667sp[1]3 + ...up to 80 terms  
var[2] := D(sp[1]2 * D(var[1], sp[1]), sp[1]);  
var[2]() := 0;
```

The plot before is:

And at the end is:

REFERENCES

- [1] Parker, G. Edgar and Sochacki, James S. Implementing the Picard iteration, *Neural, Parallel, and Scientific Computation* 4 (1996) 97-112.
- [2] Parker, G. Edgar and Sochacki, James S. A Picard-McLaurin Theorem for Initial Value PDE's, *Journal of Ordinary Differential Equations* submitted.