S function

S functions are a way to add C/C++ or any other language function to a Simulink model. These function are used to create custom blocks for very specific tasks. Most popular S function are built using the C, C++ and FORTRAN. A main driving factor for S function is that it can be saved and is transferable to another model simulation.

0.1 There are two ways to write S-function in Simulink.

1. **Legacy_Code Method of S-Function**: This method there is less control on the simulation but it is computationally very efficient and more versatile. There is no need for major changes in the pre-existing C/C++ code, here **legacy_code** creates C++ code which can be used in both Matlab and Simulink. This method is preferred for hardware simulations.

2. **Level 2 S- function**: Here the C program is written in the "wrapper file". There is more control of the S-function because simulation structure SimStruct is accessible in this method. (SimStruct is simulation data storing structure where parameters control all simulations in Sumlink model eg. simulation time, input parameters, output data type etc) . In Level-2 S function, Matlab provides processing functions which controls the SimStruct. Matlab also provides a GUI based approach for this called S-function Builder. S-function Builder method acts a GUI for the level-2 method.

0.2 Standard Variable Nomenclature

Following nomenclature must be used while declaring input, output or parameters in the S function. These variable names remains constant for all type of S functions. Note that pointers can be used in the C/C++ functions, but should be declared like an array in S function declaration. Without the nomenclature the legacy code does not compile C/C++ code.

1. Input Variables
   (a) For N input variables, syntax format is u1.....un.
   (b) For a variable array of dimension m1,m2.,syntax format is u1[m1],u2[m2].

2. Output variable
   (a) For N output variables, syntax format is y1.....yn.
   (b) For a variable array of dimension m1,m2., syntax format is y1[m1],y2[m2].

3. Parameters
   (a) For N parameters variables, syntax format is p1.....pn.
   (b) For a variable array of dimension m1,m2., syntax format is p1[m1],p2[m2].

0.3 General flow of the S function

The S function we write in any language should also follow this flow of operation. To solve a differential equation integration protocol can be coded in S function eg, Euler and RK4 . But in Level 2 S function Matlab also provide an integration function, which will execute the integration type set in the Simulink properties. In other words, for the systems involving differential equations, we can either

- Provide our own numerical integration algorithm.
- Use Matlab/Simulink integration function and choose it in the simstruct property.

Flow of S function operation:

Step 1 First step is initialization step, here the Parameters, input and output variable size allocation is done. In this step general syntax of the function and methods called is also verified.

Step 2 Next is the simulation loop, this is only executed if there are simulation time depended functions in the code. The simulation loop is independent of the initialization stage.

(a) Calculate the output of the functions for the first time step.

(b) If there is any derivative function execute this function and update the the output. To use this pass the states to the function **differentiate()** in level 2-function and in C mex function use mdl_derivative library.

```
#define MDL_DERIVATIVES
void mdlDerivatives(SimStruct *S)
```

Step 3 Repeat the Step 2 loop until simulation time is complete

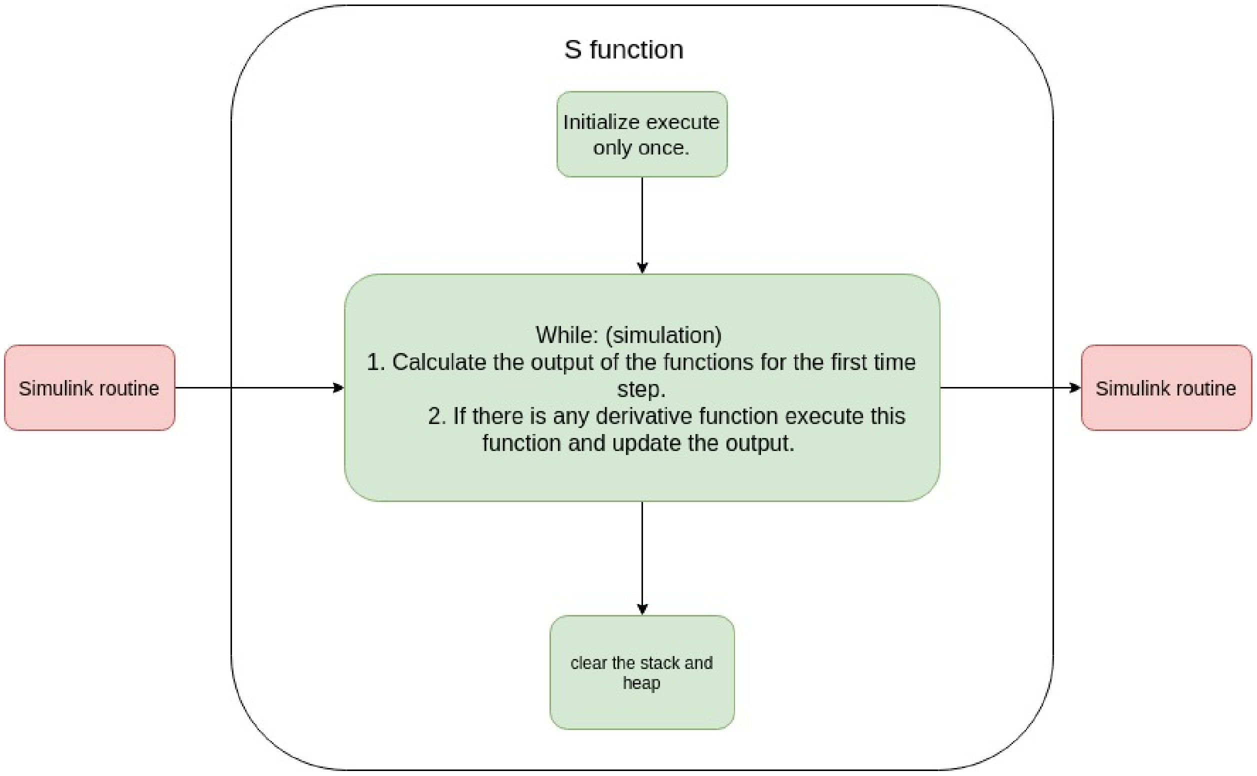

S function
Initialize execute only once.
While: (simulation)
1. Calculate the output of the functions for the first time step.
2. If there is any derivative function execute this function and update the output.
clear the stack and heap
Simulink routine
Simulink routine

Figure 1: Flow chart of the S function.

0.4 Prerequisite

Common C-compilers are as follows:

1. **Windows** C++ compiler mingw can be downloaded using the Add-On Manager in the Matlab environment. If using other MinGW installation other than Matlab i.e. Visual Studio, Codeblocks , you have set to the environment path to the Matlab, using the command.

```
setenv('MW_MINGW64_LOC',location of the file)
```

2. **Linux / Mac OS** GNU g++ Compiler installed in the system already.

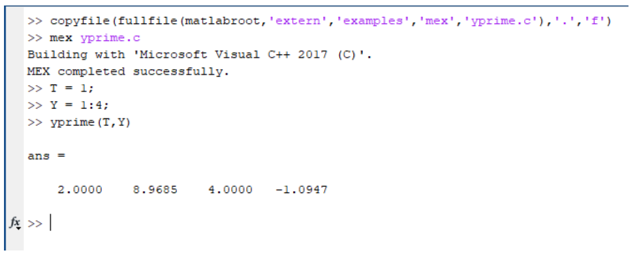
To test the installation of the C compiler, type following command in the command window

```
>> mex -setup
```

There is a build in Matlab example to test the working and and to know the working of the files called as y prime. We will use this file to test the working of the function before we write the custom function.

```
copyfile(fullfile(matlabroot,'extern','examples','mex','yprime.c'),'.','f')

%this will get the file to the current directory

>>mex yprime.c

%this command will be build the executable file using the compiler
%to test mex file in matlab

>>T=1;
>>Y=1:4;

>>yprime(T,Y)
```

the output file should look like the following file



```
>> copyfile(fullfile(matlabroot,'extern','examples','mex','yprime.c'),'.','f')
>> mex yprime.c
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
>> T = 1;
>> Y = 1:4;
>> yprime(T,Y)

ans =

    2.0000    8.9685    4.0000   -1.0947

>> 
```

Figure 2: there must be a file with extension *.mexw64 along with the *.c file, This shows that compiler is working properly in the matlab.

```c
#if !defined(MAX)
#define MAX(A, B)    ((A) > (B) ? (A) : (B))
#endif

#if !defined(MIN)
#define MIN(A, B)    ((A) < (B) ? (A) : (B))
#endif

static  double  mu = 1/82.45;
static  double  mus = 1 - 1/82.45;


static void yprime(
           double   yp[],
           double   *t,
           double   y[]
           )
{
    double  r1,r2;

    (void) t;     /* unused parameter */

    r1 = sqrt((y[0]+mu)*(y[0]+mu) + y[2]*y[2]);
    r2 = sqrt((y[0]-mus)*(y[0]-mus) + y[2]*y[2]);

    /* Print warning if dividing by zero. */
    if (r1 == 0.0 || r2 == 0.0 ){
        mexWarnMsgIdAndTxt( "MATLAB:yprime:divideByZero",
                "Division by zero!\n");
    }

    yp[0] = y[1];
    yp[1] = 2*y[3]+y[0]-mus*(y[0]+mu)/(r1*r1*r1)-mu*(y[0]-mus)/(r2*r2*r2);
    yp[2] = y[3];
    yp[3] = -2*y[1] + y[2] - mus*y[2]/(r1*r1*r1) - mu*y[2]/(r2*r2*r2);
    return;
}
```

Figure 3: Snap shot of source code yprime.c

1 Legacy_code method of S-Function

S function are the Simulink block which enables the use of the c and c++ files i.e ***.c**, ***.cpp** and supporting header files ***.h**, ***.hpp**

1.1 S Function Development Using Legacy Code

1. Develop a C/C++ code for the desired operation, make sure there is no **main()** function in the project. Also write the .h file for the function in C++ code.
2. Initialize a legacy_code development structure and declare the **Parameters**, **Inputs** and **Outputs** of the S function. Also add the parent functions (Output and Input/Initialization functions, which are C/C++ functions,to receive or output values to the simulink.), **note that parent functions declaration syntax is in m file and not C++.**
3. Build, compile legacy_code structure, which results is a wrapper .c function and .mex file.
4. Open a new Simulink workspace, add the S function block to the workspace. Rename the block in the name of the function which was created, enter the parameters value in the dialog box of the properties section.
5. Now the S-function is ready for the use. These function are reusable in other simulink models, if .c and .mex files are transferred.

1.2 Example: Step function

Now lets do the custom step saturation function which give output 1 if the any value more than 5 is given in Simulink.

1.2.1 Making mex file

1. Create two C++ file using texteditor and save in the same workspace

```
\* file my_step.c *\

#include "my_step.h"

double step(double a)
{
    double b;
    if (a>5){
        b = 1;
    }
    else{
        b=0;
    }
    return(b);

}

%%
\* file my_step.h *\
#ifndef MY_STEP_H_INCLUDED
#define MY_STEP_H_INCLUDED

double step(double a);

#endif
```

2. Create a structure that will store the properties of the .c file. using the command

```
        struc = legacy_code('initialize')
```

This is will create a structure with fields:

```
        struc =

  struct with fields:

                  SFunctionName: ''
    InitializeConditionsFcnSpec: ''
                  OutputFcnSpec: ''
                   StartFcnSpec: ''
               TerminateFcnSpec: ''
                    HeaderFiles: {}
                    SourceFiles: {}
                   HostLibFiles: {}
                 TargetLibFiles: {}
                       IncPaths: {}
                       SrcPaths: {}
                       LibPaths: {}
                     SampleTime: 'inherited'
                        Options: [1 x 1 struct]
```

Now we have to change the various attributes to the S-Function like name, sourcefile, headerfile etc

```
        struc.SFunctionName = 'my_step_block'

        %This is the name of the S Function which will be created
```

Output function: Output function is a wrapper function which take the Input from the Simulink/ Matlab and compute output from the C++ code. Here nomenclature used in the output function:

(a) $y1, y2, y3.....yn$ are used for different outputs.

(b) $u1, u2, u3.....un$ are used for different input ports.

(c) Array and pointers inputs and outputs should be called as $y1[1...n]$ **and** $x1[1....n]$

(d) $p1, p2, p3.....pn$ are used for different parameters.

In this case there is only one Input port $u1$ and one output port $y1$

```
        struc.OutputFcnSpec = 'double y1 = step(double u1)'
```

Populate the structure with the source file and the header file. To add multiple .h and .c use the same struct and declare

```
        struc.HeaderFiles = {'my_step.h'}
        struc.SourceFiles={'my_step.c'}
```

Now the final struc will look like the following.

```
      struc =

  struct with fields:

                  SFunctionName: 'my_step_block'
    InitializeConditionsFcnSpec: ''
                  OutputFcnSpec: 'double y1 = step(double u1)'
                   StartFcnSpec: ''
               TerminateFcnSpec: ''
                    HeaderFiles: {'my_step.h'}
                    SourceFiles: {'my_step.c'}
                   HostLibFiles: {}
                 TargetLibFiles: {}
                       IncPaths: {}
                       SrcPaths: {}
                       LibPaths: {}
                     SampleTime: 'inherited'
                        Options: [1 x 1 struct]
```

3. Create the .c and compile this code using the legacy_code compile statement as follows

   ```
   legacy_code('sfcn_cmex_generate',struc)
   legacy_code('compile',struc)
   ```

   Matlab will create two files in the workspace a .mex file and a .c file, these two files are used to develop S-function in simulink and also these files are transferable to other directory.

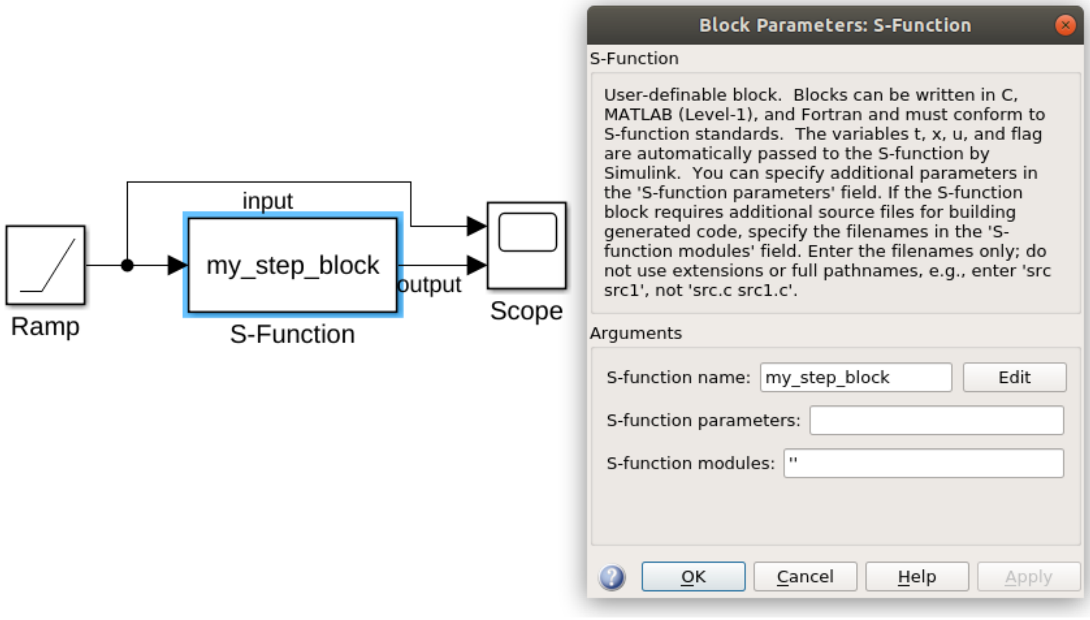
4. Loading an S function in Simulink model

   (a) Add S function block from simulink library browser.

   (b) Rename the block from the default name - **system** to the name **name of the block**

   (c) Output should look like the following figure 5

   Now add the function with the input test signal and run the simulation. The results should be as follows:



Block Parameters: S-Function

S-Function

User-definable block. Blocks can be written in C, MATLAB (Level-1), and Fortran and must conform to S-function standards. The variables t, x, u, and flag are automatically passed to the S-function by Simulink. You can specify additional parameters in the 'S-function parameters' field. If the S-function block requires additional source files for building generated code, specify the filenames in the 'S-function modules' field. Enter the filenames only; do not use extensions or full pathnames, e.g., enter 'src src1', not 'src.c src1.c'.

Arguments

S-function name: my_step_block

S-function parameters:

S-function modules: ''

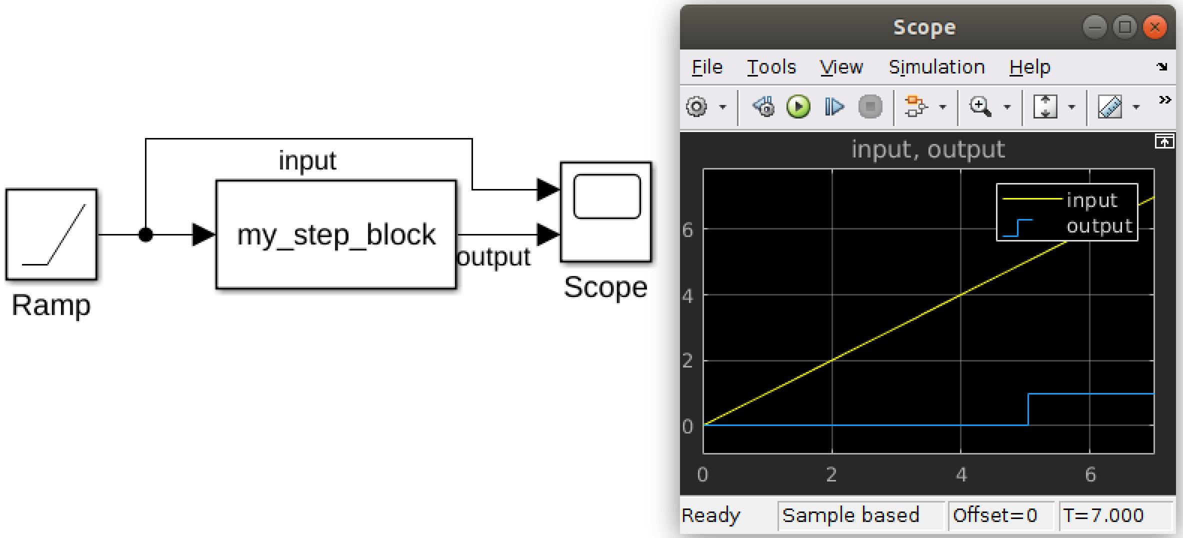
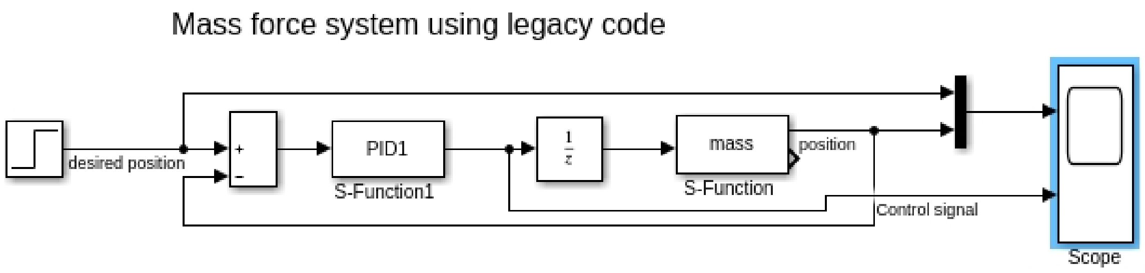


Figure 5: Simulation output of the block

1.3 Example: Spring mass force system using PID controller

Figure 6: Mass force system built using S-functions



1. C++ code for the mass force dynamics.

```
#include "math.h"
#include "massc.h"
void mass(double u, double K, double C,double M, double delta_t, double *y1, double *y2)
{
// u - input , K,C,t - parameters, y1 and y2 are pointers for the output.
// U - control input
// K, C are spring constant and damping
// t - euler time step
//x_dot is the velocity and acceleration
// new_x is the updated velcoty and distance
double x_dot[2];
static double new_x[2] = {0.0,0.0};
x_dot[0] = new_x[1];
x_dot[1] = (1/M) * (u - K * new_x[0] - C * new_x[1]);
new_x[0] = euler(x_dot[0],new_x[0],delta_t);
new_x[1] = euler(x_dot[1],new_x[1],delta_t);
*y1 = new_x[0];
*y2 = new_x[1];
}



double euler(double y,double x,double delta_t)
{
double z = y * delta_t + x;
return z;
}
```

H File

```
#ifndef MASSC
#define MASSC
double euler(double y,double x,double delta_t);
void mass(double u, double K, double C,double M, double delta_t, double *y1, double *y2);
#endif
```

For the this C++ code the parameters are K,C,M and delay time (Euler time and step). Input is u and output pointer are y1 and y2. The code is given below.Here, Euler method of differentiation is used and other more complex methods can be used.

2. C++ code for the PID control.

```
%%%% file PIDC.cpp
#include "PIDC.h"
#include "math.h"
double pid (double error ,double Kp ,double Ki,double Kd)
{
static double previous_error = 0;
static double error_integral = 0.0;
// error - input; P,I,D paramters
double P_out,I_out,D_out,PIDOut;
P_out = error * Kp;
error_integral = error_integral + error;
I_out = error_integral * Ki;
D_out = (error-previous_error) * Kd;
PIDOut = P_out + I_out + D_out;
previous_error = error;
return PIDOut;
}
```

H file

```
    %%%% file PIDC.h
#ifndef PIDC
#define PIDC
#include "iostream"
double pid (double error ,double Kp ,double Ki,double Kd);
#endif
```

For the this C++ code the parameters are P,I and D. Input is error and output is out.

3. Build and compile the Mass and PID controller code using the legacy_code. The following code is used to create the mex code for the PID and mass block.

```
massstruct = legacy_code('initialize');
massstruct.SFunctionName = 'mass';
massstruct.SourceFiles = {'massc.cpp'};
massstruct.HeaderFiles = {'massc.h'};
massstruct.OutputFcnSpec =  'void mass(double u1, double p1, double p2, double p3, double p4, doubl
pidstruct  = legacy_code('initialize');
pidstruct.SFunctionName = 'PID1';
pidstruct.SourceFiles = {'PIDC.cpp'};
pidstruct.HeaderFiles={'PIDC.h'};
pidstruct.OutputFcnSpec = 'double y1 = pid(double u1, double p1, double p2, double p3)';
legacy_code('sfcn_cmex_generate',massstruct);
legacy_code('compile',massstruct);
legacy_code('sfcn_cmex_generate',pidstruct);
legacy_code('compile',pidstruct);
```

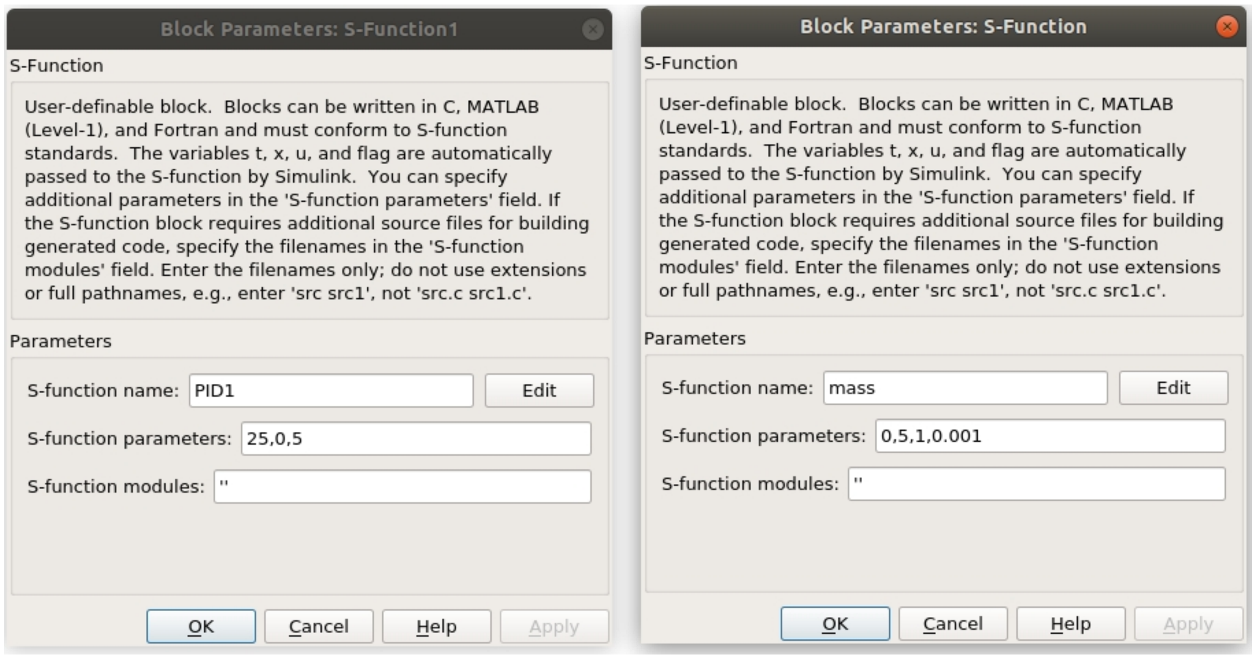
4. In Simulink, add 2 S-function blocks to the workspace and in the system change the block parameters as the figure shown below
Note: Parameters order is declared in cpp file and the legacy_code struct as indicated below.
Mass force dynamics parameters order of declaration is as follows.

```
void mass(double u, double K, double C,double M, double delta_t, double *y1, double *y2);
massstruct.OutputFcnSpec =  'void mass(double u1, double p1, double p2, double p3, double p4, doubl
// Here parameters are K is p1, C is p2, M is p3, delta_t is p4.
```

PID controller parameters order of decleration are as follows.

```
double pid (double error ,double Kp ,double Ki,double Kd);
pidstruct.OutputFcnSpec = 'double y1 = pid(double u1, double p1, double p2, double p3)';
// Here parameters are Kp is p1, Ki is p2, Kd is p3.
```

Based on these order the parameters are populated.



Block Parameters: S-Function1

S-Function

User-definable block. Blocks can be written in C, MATLAB (Level-1), and Fortran and must conform to S-function standards. The variables t, x, u, and flag are automatically passed to the S-function by Simulink. You can specify additional parameters in the 'S-function parameters' field. If the S-function block requires additional source files for building generated code, specify the filenames in the 'S-function modules' field. Enter the filenames only; do not use extensions or full pathnames, e.g., enter 'src src1', not 'src.c src1.c'.

Parameters

S-function name: PID1

S-function parameters: 25,0,5

S-function modules: ''

Block Parameters: S-Function

S-Function

User-definable block. Blocks can be written in C, MATLAB (Level-1), and Fortran and must conform to S-function standards. The variables t, x, u, and flag are automatically passed to the S-function by Simulink. You can specify additional parameters in the 'S-function parameters' field. If the S-function block requires additional source files for building generated code, specify the filenames in the 'S-function modules' field. Enter the filenames only; do not use extensions or full pathnames, e.g., enter 'src src1', not 'src.c src1.c'.

Parameters

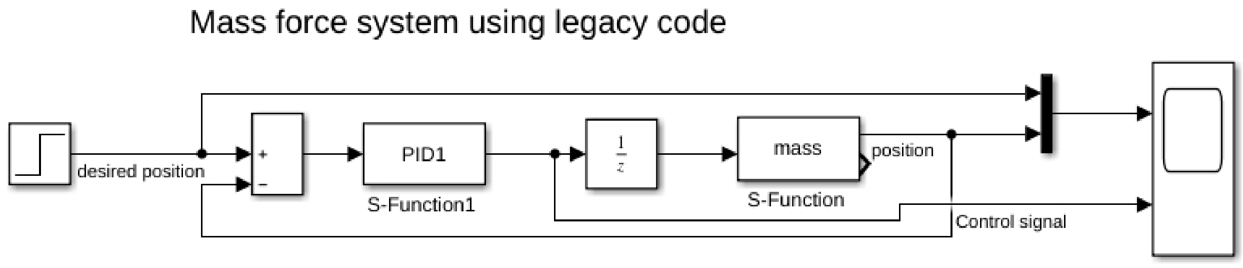
S-function name: mass

S-function parameters: 0,5,1,0.001

S-function modules: ''

Figure 7: block parameters for the S-function block.

5. Construct the Simulink model as show in the fig 8. Since Custom numerical integration is provided and time step of that is 0.001 add a unit delay block with identical delay. Note is the unit delay block is not added result will go infinity and simulink is display an error.



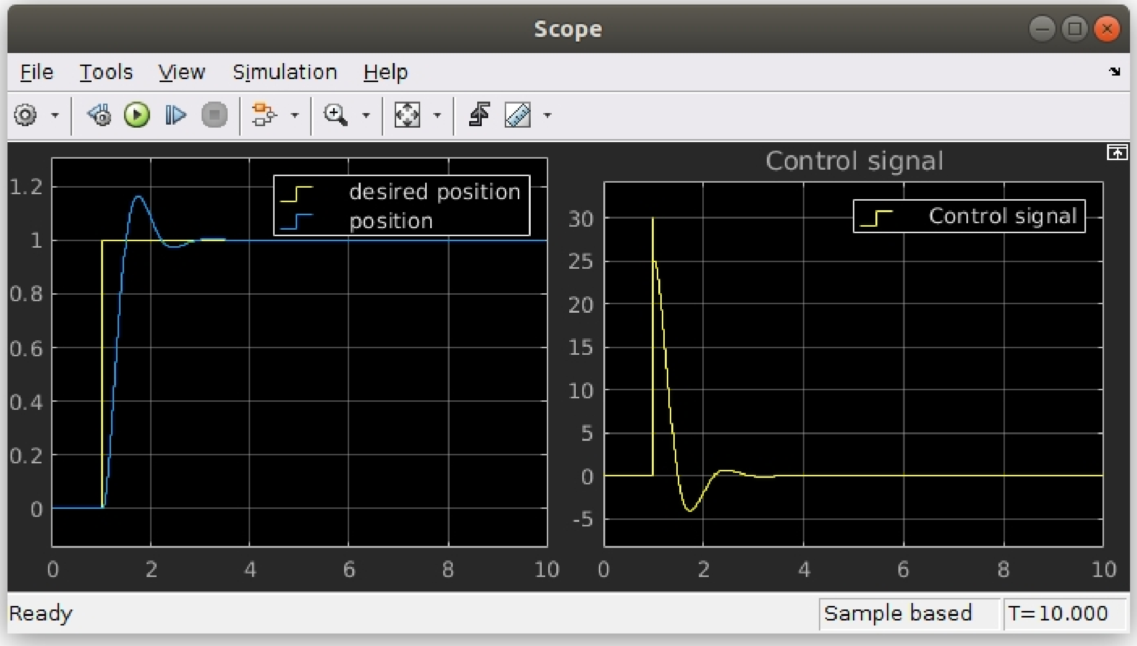


Figure 8: Results.

2 S Function Builder

This is a built in Simulink block which provides a GUI for **parameters, Input and Output** variable and its size declarations. **Simulink builder** not only has all the advantages of that of legacy_code but also provides initialization parameters (states), whose value can be initialized in the beginning of code and can dynamically changed. **S function builder** is a GUI over the level-2 S function, So this builder will create a wrapper.c file where the c code can be changed. S function builder follows the syntax and function calling method similar to that of the level 2 s function.

2.1 Steps for using S function builder in simulink

1. Copy the S Function Builder from Simulink toolbox into the **Simulink model**.
2. Double click on block to open the parameters windows as shown in the figure.
3. Now there are 9 tabs window, where the characteristics of the S function can be changed. *Declaration that needs to be done in this section is in the C programming syntax and pointers can be used*

(a) **Initialize:** Here the variable are discrete and continuous states are declared. These states values can be stored and used in different time step of simulation.
To access discrete states we have to use, xD[0 ... 10]
To access continuous states we have to use xC[0 ... 10]
Also note do not use xd use xD.

(b) **Data properties:** In this sections we declare the input, output and parameters size and data type. Input output and parameters can be declared as array and also multiple input and output for a block can be declared here.

(c) **Libraries:** Here the header files and the source files are added. Also the functions used in the source file and not declared in the .h file should be declared in the dialog box given.

(d) **Start:** Here the states are declared and initialized. Note that this part of code will run once at the start of the simulation.

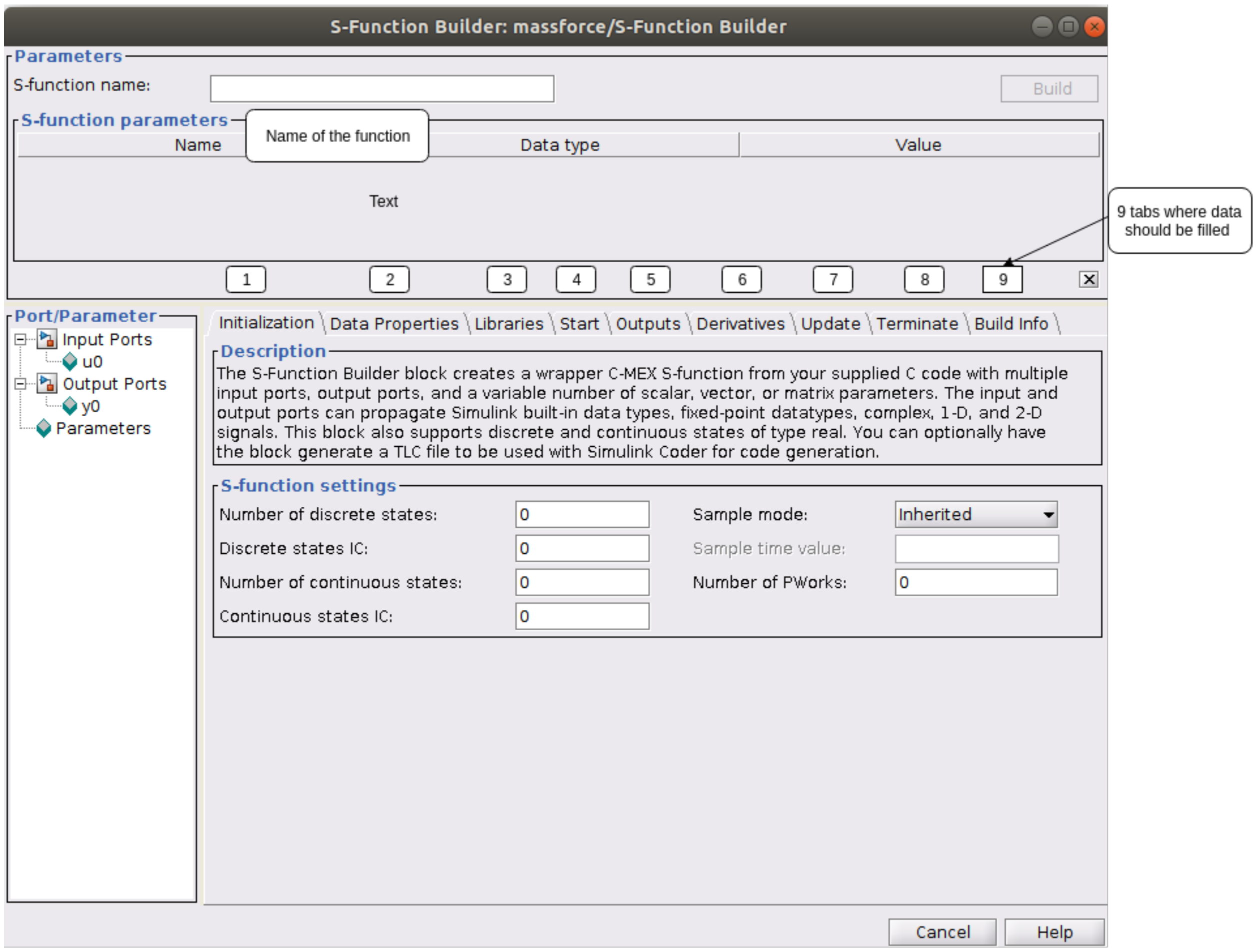
(e) **Output:** Here the output functions are declared, the output functions are the ones which are responsible for the output y1[0],y1[0] .... y1[n] , the code or algorithm which needs to run will be called in this function. Making this main() or (parent) function and all the functions declared in the libraries section are nested functions. To access input variables enable the check mark at the bottom. Function calling the other files can be coded here.

(f) **Differentiate:** If any state needs to differentiated it will be declared here.

(g) **Update:** If states need to be update after each simulation time, then state with update condition should be declared here.

(h) **Terminate:** Here the part of code that need to run at the end of all the simulation, is declared eg. freeing up the memory in the ram etc.

(i) **Build info:** When the S function is built, this section will provide the information of the C files which are created.


S-Function Builder: massforce/S-Function Builder

Parameters

S-function name:

S-function parameters

Name	Data type	Value

Port/Parameter
- Input Ports
  - u0
- Output Ports
  - y0
- Parameters

Initialization | Data Properties | Libraries | Start | Outputs | Derivatives | Update | Terminate | Build Info

Description

The S-Function Builder block creates a wrapper C-MEX S-function from your supplied C code with multiple input ports, output ports, and a variable number of scalar, vector, or matrix parameters. The input and output ports can propagate Simulink built-in data types, fixed-point datatypes, complex, 1-D, and 2-D signals. This block also supports discrete and continuous states of type real. You can optionally have the block generate a TLC file to be used with Simulink Coder for code generation.

S-function settings

Number of discrete states:	0	Sample mode:	Inherited
Discrete states IC:	0	Sample time value:	
Number of continuous states:	0	Number of PWorks:	0
Continuous states IC:	0		

Figure 9: (9 windows where data must be populated.

4. When S function is build, it will create **two *.C files, one .TLC files** and **one complied S function file**. One of the C file created is a wrapper file where the function can be called, another is complete S function code in level 2 S function format. If the properties of the function is changed in the file must be built again.

2.2 Example: Mass force system with PID controller

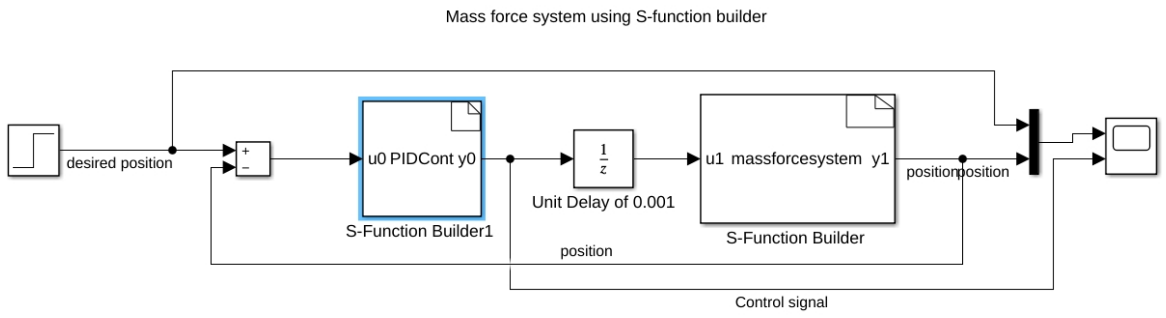


Figure 10: Mass force system

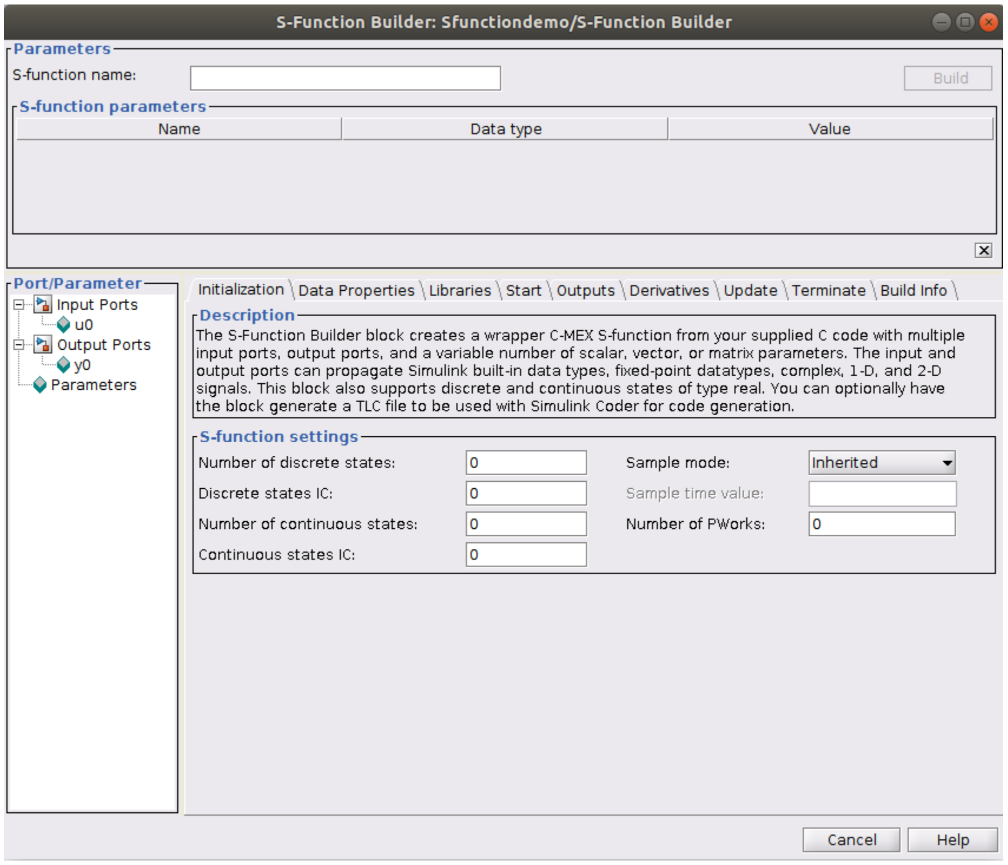
1. **C/C++ files Mass dynamics system files**

```
%%%%%%%%%%% file name massc.cpp %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#include "math.h"
#include "massc.h"

double mass(double *delta_t ,double u1, double *K, double *C, double *M)
{
    // u - input , K,C,t - parameters
    // U - control input
    // K, C are spring constant and damping
    // delta_t - euler time step
    //x_dot is the velocity and acceleration
    //new_x[0] in the distance
    double x_dot[2];
    static double new_x[2] = {0.0,0.0};
    // Note: All the paramters must be pointers, Input and state are variables(double),
    x_dot[1] = (1/ *M) * ( u1  - *K * new_x[0] - *C * new_x[1]);
    x_dot[0] = new_x[1];
    new_x[0] = euler(x_dot[0],new_x[0], *delta_t);
    new_x[1]  = euler(x_dot[1],new_x[1], *delta_t);
    return new_x[0];
}

double euler(double y,double x,double delta_t)
{
    double z = y * delta_t + x;
    return z;
}



%%%%%%%%%%%% file name massc.h %%%%%%%%%%%%
#ifndef MASSC
#define MASSC

double euler(double y,double x,double t);
double mass(double *delta_t, double u1, double *K, double *C, double *M);
#endif
```

PID controller files

```
%%%%%%%% file name PIDC.cpp %%%%%%%%%%%%%%%%%
#include "PIDC.h"
#include "math.h"

double pid (double error ,double *Kp ,double *Ki,double *Kd)
{
    // error - input P,I,D paramters
    static double previous_error = 0;
    static double error_integral = 0.0;
    // Note: all the paramters should be double variable.
    double P_out,I_out, D_out, Out;
    P_out  = error * *Kp;
    I_out =  error_integral * *Ki;
    D_out = (error-previous_error) * *Kd;
    Out = P_out +  I_out + D_out;
    //update
    previous_error = error;
    error_integral = error + error_integral;
    return Out;
}
%%%%%%%%% file name PIDC.h %%%%%%%%%%%%%%%%%%
#ifndef PID
#define PID
#include "iostream"
double pid (double error ,double *Kp ,double *Ki,double *Kd);
#endif
```

2. Load the S function builder to the Simulink workspace. Double click the function to get the properties in the GUI or also called as function builder. As shown in the fig


S-Function Builder: Sfunctiondemo/S-Function Builder

Parameters

S-function name:

Build

S-function parameters

Name	Data type	Value

Port/Parameter
- Input Ports
  - u0
- Output Ports
  - y0
- Parameters

Initialization | Data Properties | Libraries | Start | Outputs | Derivatives | Update | Terminate | Build Info

Description

The S-Function Builder block creates a wrapper C-MEX S-function from your supplied C code with multiple input ports, output ports, and a variable number of scalar, vector, or matrix parameters. The input and output ports can propagate Simulink built-in data types, fixed-point datatypes, complex, 1-D, and 2-D signals. This block also supports discrete and continuous states of type real. You can optionally have the block generate a TLC file to be used with Simulink Coder for code generation.

S-function settings

Number of discrete states:	0	Sample mode:	Inherited
Discrete states IC:	0	Sample time value:	
Number of continuous states:	0	Number of PWorks:	0
Continuous states IC:	0		

Cancel Help

Figure 11: Fill dialog box of the new block

3. Name the S-funtion as **massforcesystem**

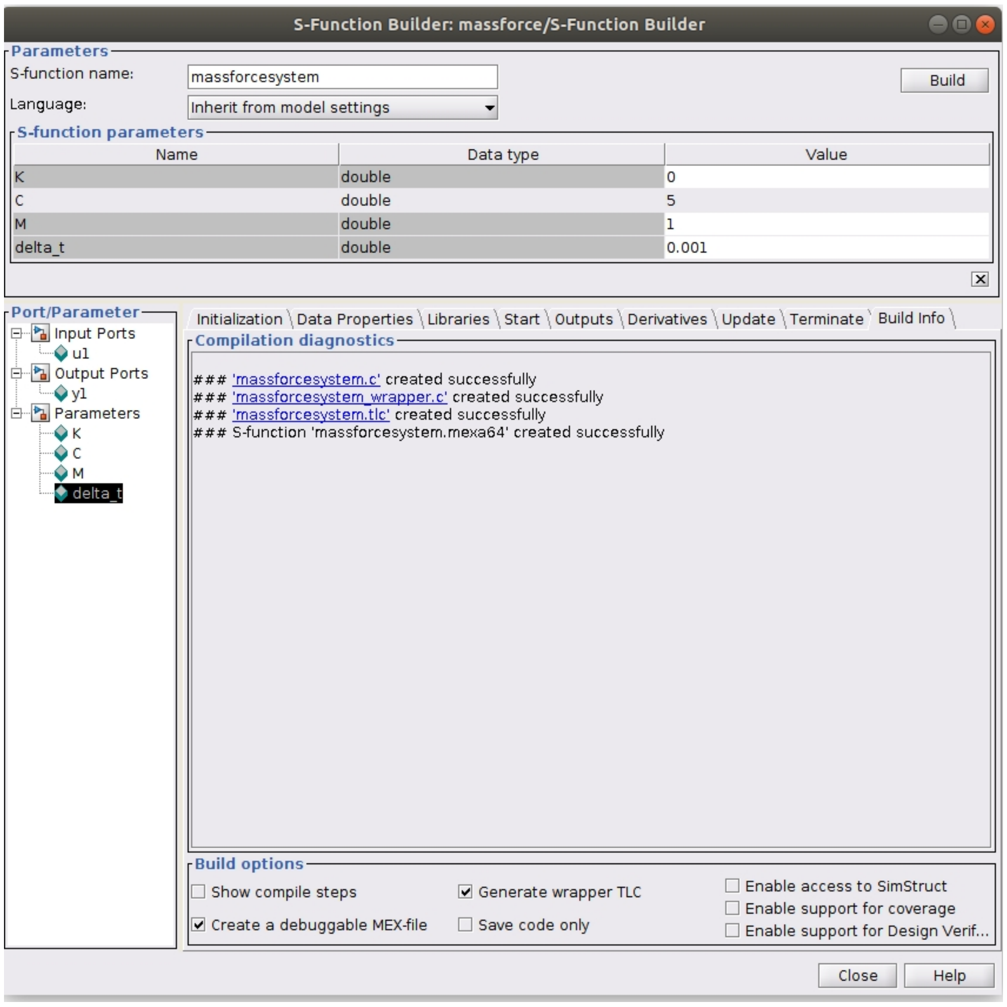
4. In the Data *Properties* → *Inputports* → *add or rename the port* similarly follow the name procedure. For the system parameters go the *DataProperties* → *Parameters* add parameters K,C,m,and delta_t.

5. In the Libraries tab in the GUI. Add the Source file, **massc.cpp** and in the Header file **include "massc.h"**.

6. Now in the output tab, call the function.

```
y1[0] = mass( (double*)delta_t,u1[0], (double*) K,  (double*)C,  (double*)M);
```

7. In the build Info tab, check "Generation wrapper TLC". and Build the system, Now there will be a *_wrapper.c, *.c and *.tlc file in the workdirectory. the TLC file is transferable.



Name	Data type	Value
K	double	0
C	double	5
M	double	1
delta_t	double	0.001

```
### 'massforcesystem.c' created successfully
### 'massforcesystem_wrapper.c' created successfully
### 'massforcesystem.tlc' created successfully
### S-function 'massforcesystem.mexa64' created successfully
```

Build options: [ ] Show compile steps; [✓] Create a debuggable MEX-file; [✓] Generate wrapper TLC; [ ] Save code only; [ ] Enable access to SimStruct; [ ] Enable support for coverage; [ ] Enable support for Design Verif...

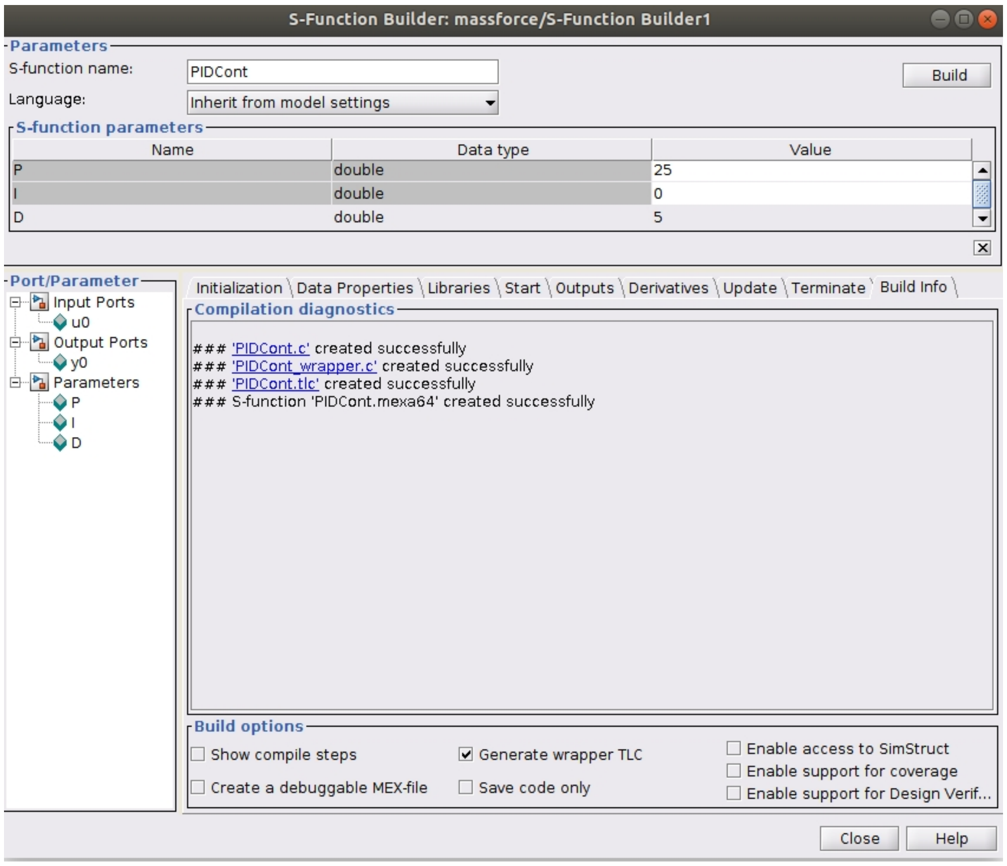
Figure 12: Output of the dialog box after building the function.

8. Open another S-function block in the block for PID controller.

9. In the parameters section in the "data parameters" add **P,I and D**.

10. In the header file section of libraries add **include "PIDC.h"** and in the source file section add the **PIDC.cpp** file.

11. Build the function in the output of that function will look like fig13


S-Function Builder: massforce/S-Function Builder1

Parameters

S-function name: PIDCont

Language: Inherit from model settings

S-function parameters

Name	Data type	Value
P	double	25
I	double	0
D	double	5

Port/Parameter

- Input Ports
  - u0
- Output Ports
  - y0
- Parameters
  - P
  - I
  - D

Compilation diagnostics

```
### 'PIDCont.c' created successfully
### 'PIDCont_wrapper.c' created successfully
### 'PIDCont.tlc' created successfully
### S-function 'PIDCont.mexa64' created successfully
```

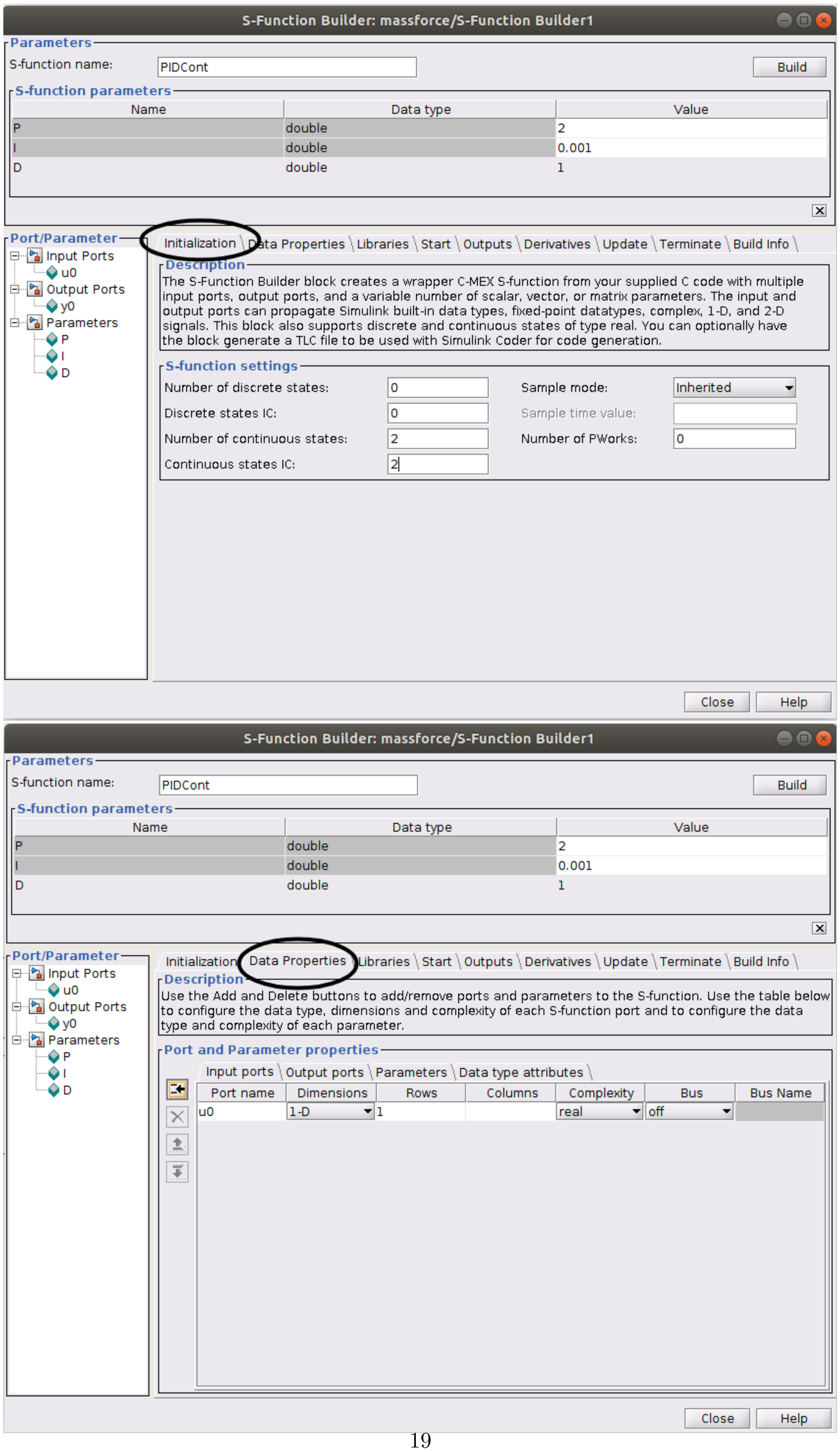
Build options

- [ ] Show compile steps
- [ ] Create a debuggable MEX-file
- [✓] Generate wrapper TLC
- [ ] Save code only
- [ ] Enable access to SimStruct
- [ ] Enable support for coverage
- [ ] Enable support for Design Verif...

Figure 13: Pid controller S function output.

Figure 14: Output of the mass force system with PID controller

Pictorial example of data populated S function builder.



S-Function Builder: massforce/S-Function Builder1

Parameters

S-function name: PIDCont

S-function parameters

Name	Data type	Value
P	double	2
I	double	0.001
D	double	1

Port/Parameter
- Input Ports
  - u0
- Output Ports
  - y0
- Parameters
  - P
  - I
  - D

Initialization | Data Properties | Libraries | Start | Outputs | Derivatives | Update | Terminate | Build Info

Description

The S-Function Builder block creates a wrapper C-MEX S-function from your supplied C code with multiple input ports, output ports, and a variable number of scalar, vector, or matrix parameters. The input and output ports can propagate Simulink built-in data types, fixed-point datatypes, complex, 1-D, and 2-D signals. This block also supports discrete and continuous states of type real. You can optionally have the block generate a TLC file to be used with Simulink Coder for code generation.

S-function settings

Number of discrete states:	0	Sample mode:	Inherited
Discrete states IC:	0	Sample time value:	
Number of continuous states:	2	Number of PWorks:	0
Continuous states IC:	2		

S-Function Builder: massforce/S-Function Builder1

Parameters

S-function name: PIDCont

S-function parameters

Name	Data type	Value
P	double	2
I	double	0.001
D	double	1

Port/Parameter
- Input Ports
  - u0
- Output Ports
  - y0
- Parameters
  - P
  - I
  - D

Initialization | Data Properties | Libraries | Start | Outputs | Derivatives | Update | Terminate | Build Info

Description

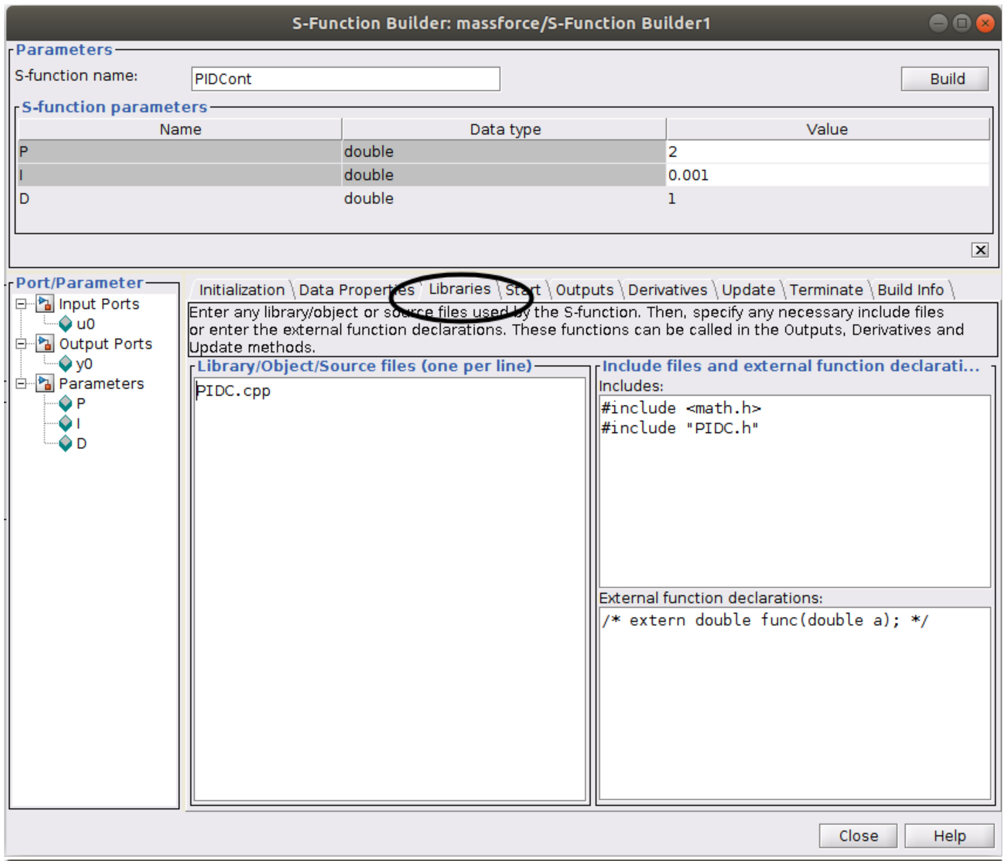
Use the Add and Delete buttons to add/remove ports and parameters to the S-function. Use the table below to configure the data type, dimensions and complexity of each S-function port and to configure the data type and complexity of each parameter.

Port and Parameter properties

Input ports | Output ports | Parameters | Data type attributes

Port name	Dimensions	Rows	Columns	Complexity	Bus	Bus Name
u0	1-D	1		real	off	

Figure 15: First is the Initialization tab and Second is the Data properties tab


S-Function Builder: massforce/S-Function Builder1

Parameters

S-function name: PIDCont

S-function parameters

Name	Data type	Value
P	double	2
I	double	0.001
D	double	1

Port/Parameter: Input Ports u0; Output Ports y0; Parameters P, I, D

Initialization \ Data Properties \ Libraries \ Start \ Outputs \ Derivatives \ Update \ Terminate \ Build Info

Enter any library/object or source files used by the S-function. Then, specify any necessary include files or enter the external function declarations. These functions can be called in the Outputs, Derivatives and Update methods.

Library/Object/Source files (one per line)

```
PIDC.cpp
```

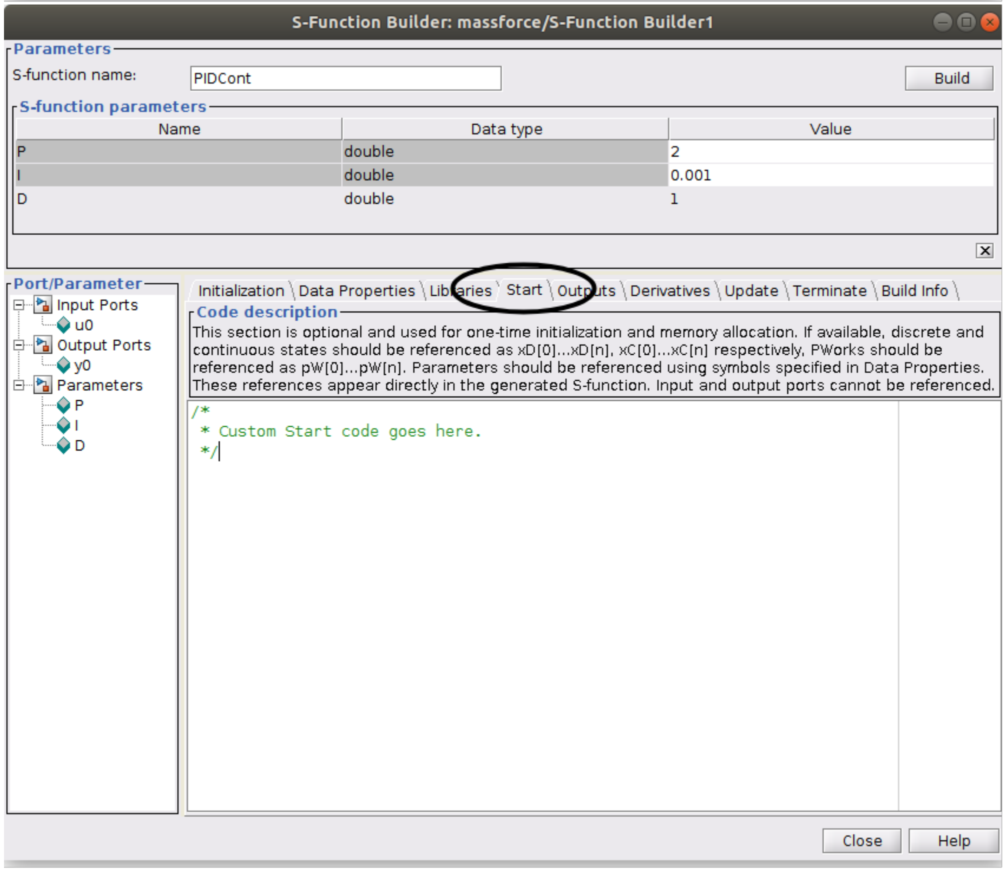
Include files and external function declarati...

Includes:

```
#include <math.h>
#include "PIDC.h"
```

External function declarations:

```
/* extern double func(double a); */
```


S-Function Builder: massforce/S-Function Builder1

Parameters

S-function name: PIDCont

S-function parameters

Name	Data type	Value
P	double	2
I	double	0.001
D	double	1

Port/Parameter: Input Ports u0; Output Ports y0; Parameters P, I, D

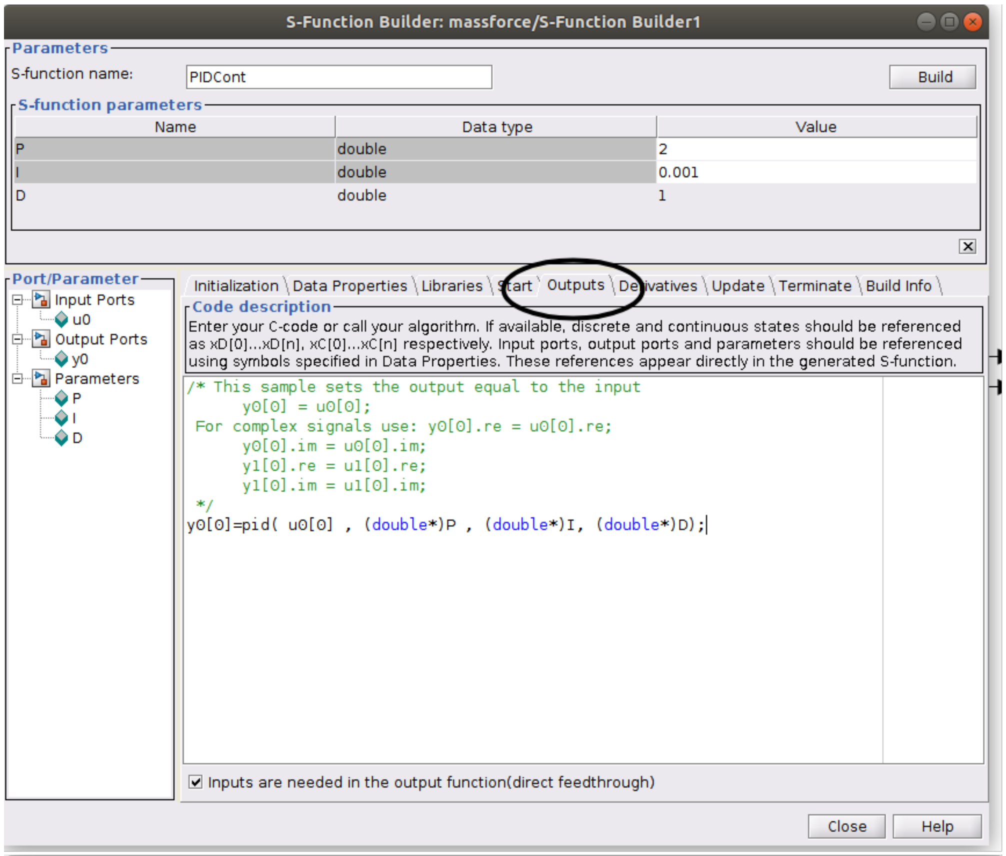
Initialization \ Data Properties \ Libraries \ Start \ Outputs \ Derivatives \ Update \ Terminate \ Build Info

Code description

This section is optional and used for one-time initialization and memory allocation. If available, discrete and continuous states should be referenced as xD[0]...xD[n], xC[0]...xC[n] respectively, PWorks should be referenced as pW[0]...pW[n]. Parameters should be referenced using symbols specified in Data Properties. These references appear directly in the generated S-function. Input and output ports cannot be referenced.

```
/*
 * Custom Start code goes here.
 */
```

Figure 16: Libraries and Start tab.



S-Function Builder: massforce/S-Function Builder1

Parameters

S-function name: PIDCont

S-function parameters

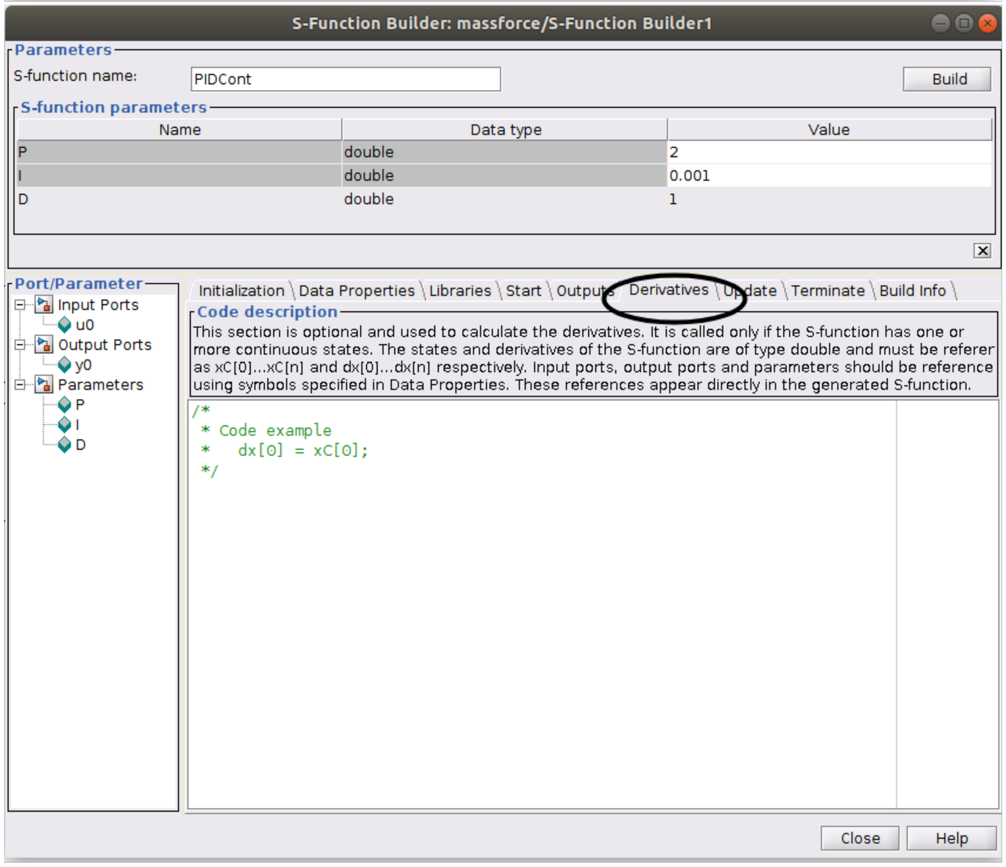
Name	Data type	Value
P	double	2
I	double	0.001
D	double	1

Code description

Enter your C-code or call your algorithm. If available, discrete and continuous states should be referenced as xD[0]...xD[n], xC[0]...xC[n] respectively. Input ports, output ports and parameters should be referenced using symbols specified in Data Properties. These references appear directly in the generated S-function.

```
/* This sample sets the output equal to the input
      y0[0] = u0[0];
 For complex signals use: y0[0].re = u0[0].re;
      y0[0].im = u0[0].im;
      y1[0].re = u1[0].re;
      y1[0].im = u1[0].im;
 */
y0[0]=pid( u0[0] , (double*)P , (double*)I, (double*)D);
```

☑ Inputs are needed in the output function(direct feedthrough)



S-Function Builder: massforce/S-Function Builder1

Parameters

S-function name: PIDCont

S-function parameters

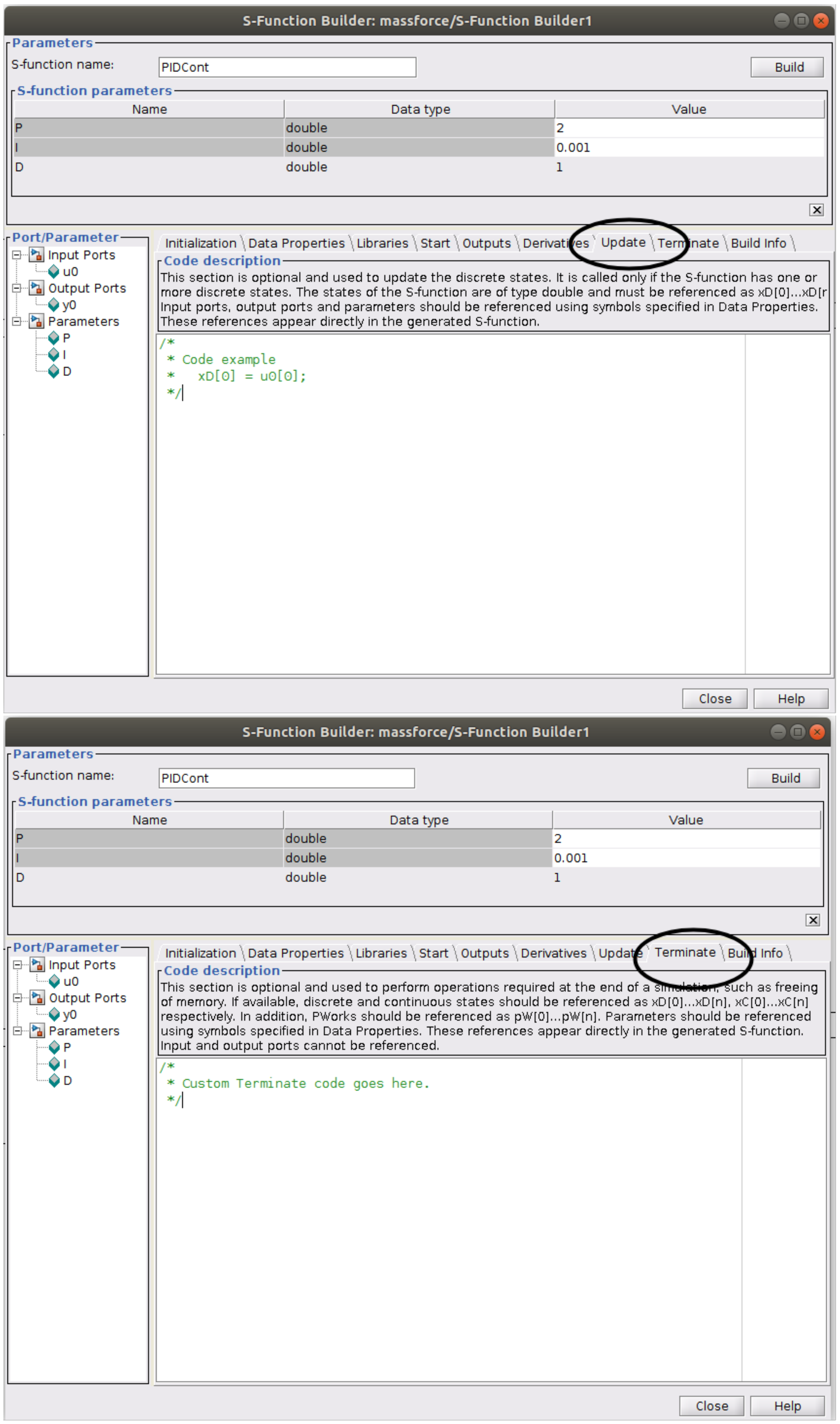
Name	Data type	Value
P	double	2
I	double	0.001
D	double	1

Code description

This section is optional and used to calculate the derivatives. It is called only if the S-function has one or more continuous states. The states and derivatives of the S-function are of type double and must be referer as xC[0]...xC[n] and dx[0]...dx[n] respectively. Input ports, output ports and parameters should be reference using symbols specified in Data Properties. These references appear directly in the generated S-function.

```
/*
 * Code example
 *   dx[0] = xC[0];
 */
```

Figure 17: Output and Derivative tab



S-function name: PIDCont

Name	Data type	Value
P	double	2
I	double	0.001
D	double	1

Code description

This section is optional and used to update the discrete states. It is called only if the S-function has one or more discrete states. The states of the S-function are of type double and must be referenced as xD[0]...xD[r Input ports, output ports and parameters should be referenced using symbols specified in Data Properties. These references appear directly in the generated S-function.

```
/*
 * Code example
 *   xD[0] = u0[0];
 */
```

S-function name: PIDCont

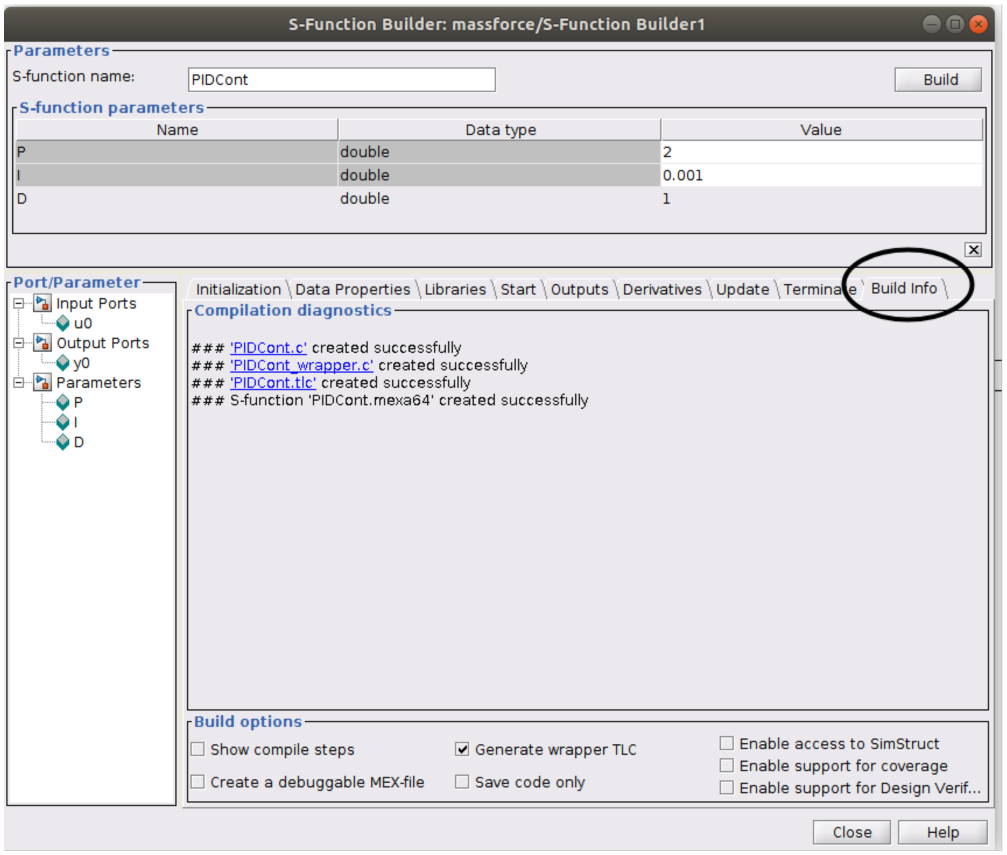
Name	Data type	Value
P	double	2
I	double	0.001
D	double	1

Code description

This section is optional and used to perform operations required at the end of a simulation, such as freeing of memory. If available, discrete and continuous states should be referenced as xD[0]...xD[n], xC[0]...xC[n] respectively. In addition, PWorks should be referenced as pW[0]...pW[n]. Parameters should be referenced using symbols specified in Data Properties. These references appear directly in the generated S-function. Input and output ports cannot be referenced.

```
/*
 * Custom Terminate code goes here.
 */
```

Figure 18: Update and Terminate tab


S-Function Builder: massforce/S-Function Builder1

Parameters

S-function name: PIDCont

S-function parameters

Name	Data type	Value
P	double	2
I	double	0.001
D	double	1

Compilation diagnostics

```
### 'PIDCont.c' created successfully
### 'PIDCont_wrapper.c' created successfully
### 'PIDCont.tlc' created successfully
### S-function 'PIDCont.mexa64' created successfully
```

Build options

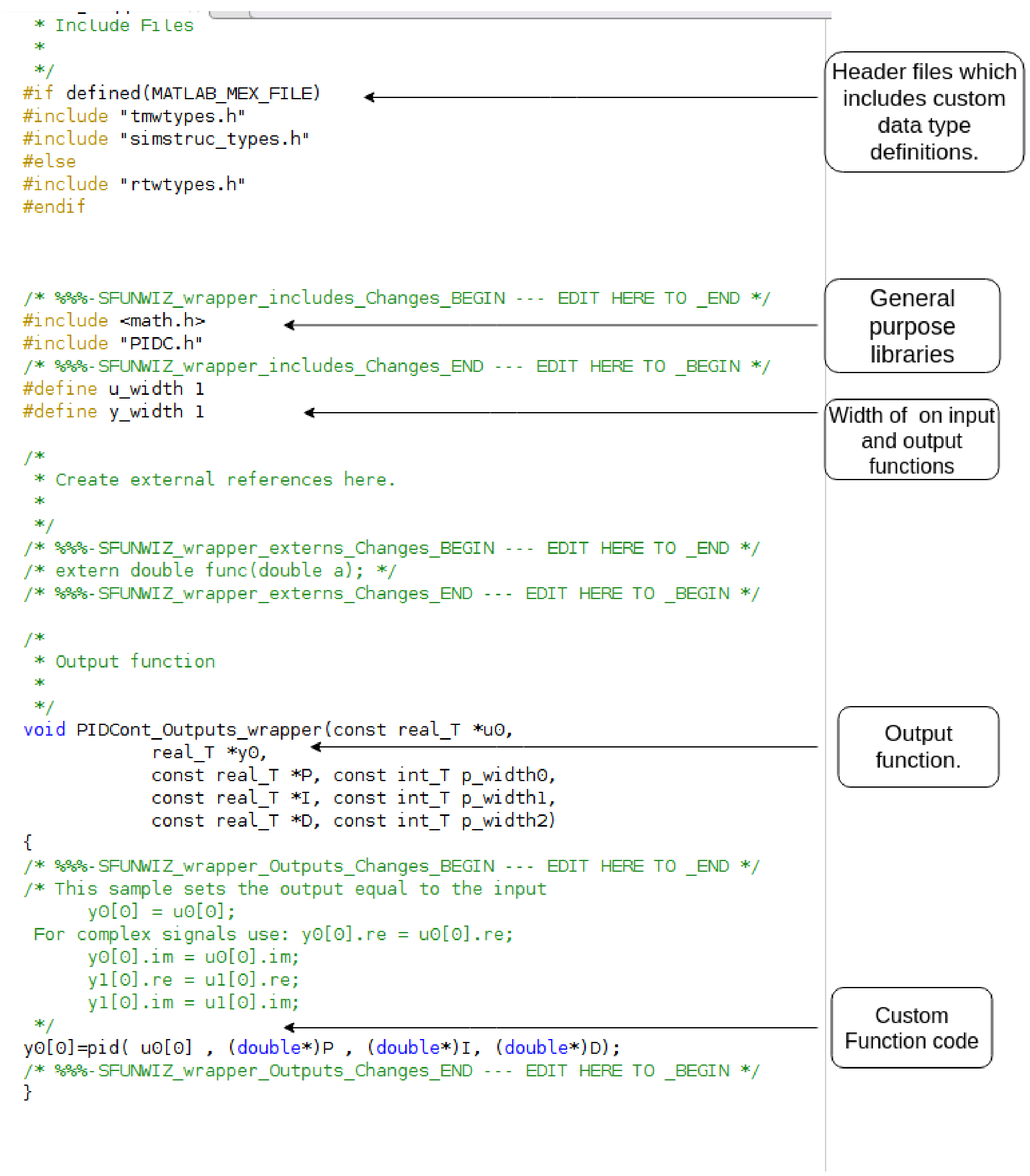
- ☐ Show compile steps
- ☐ Create a debuggable MEX-file
- ☑ Generate wrapper TLC
- ☐ Save code only
- ☐ Enable access to SimStruct
- ☐ Enable support for coverage
- ☐ Enable support for Design Verif...

Figure 19: Build Info tab

[H]

Example of the wrapper.c and function.c files created my Matlab.

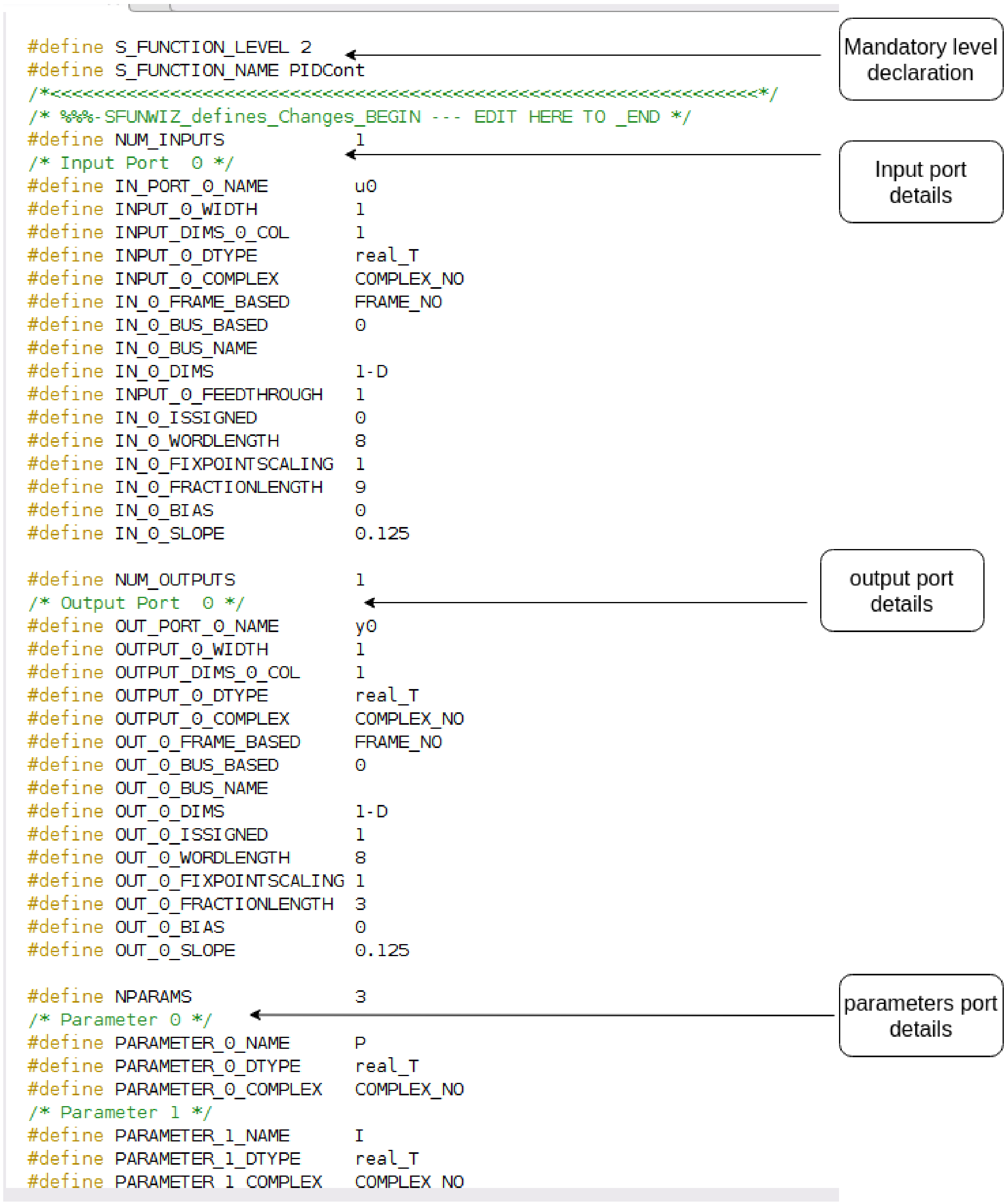
Wrapper.c file



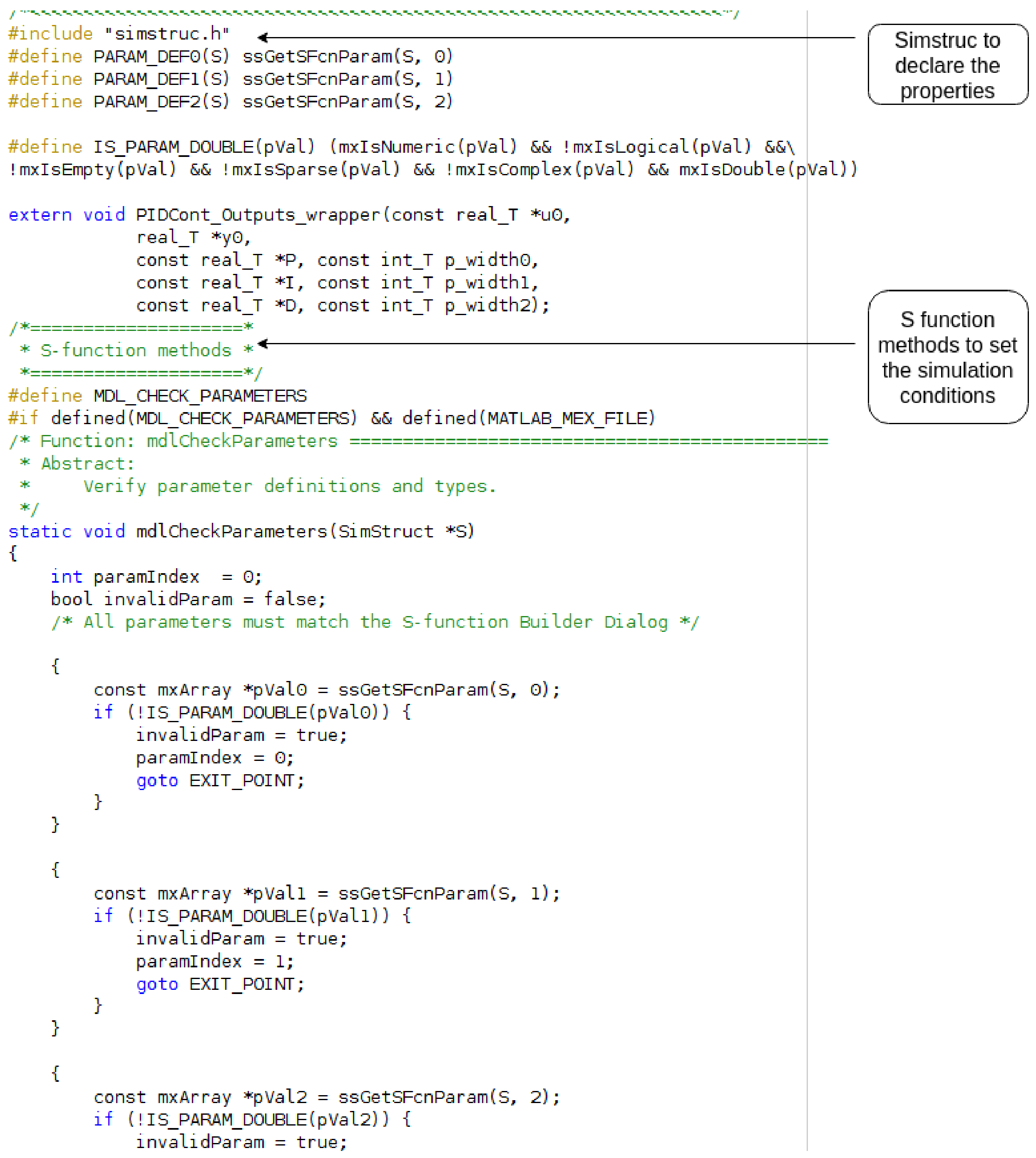
```
 * Include Files
 *
 */
#if defined(MATLAB_MEX_FILE)
#include "tmwtypes.h"
#include "simstruc_types.h"
#else
#include "rtwtypes.h"
#endif



/* %%%-SFUNWIZ_wrapper_includes_Changes_BEGIN --- EDIT HERE TO _END */
#include <math.h>
#include "PIDC.h"
/* %%%-SFUNWIZ_wrapper_includes_Changes_END --- EDIT HERE TO _BEGIN */
#define u_width 1
#define y_width 1

/*
 * Create external references here.
 *
 */
/* %%%-SFUNWIZ_wrapper_externs_Changes_BEGIN --- EDIT HERE TO _END */
/* extern double func(double a); */
/* %%%-SFUNWIZ_wrapper_externs_Changes_END --- EDIT HERE TO _BEGIN */

/*
 * Output function
 *
 */
void PIDCont_Outputs_wrapper(const real_T *u0,
            real_T *y0,
            const real_T *P, const int_T p_width0,
            const real_T *I, const int_T p_width1,
            const real_T *D, const int_T p_width2)
{
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_BEGIN --- EDIT HERE TO _END */
/* This sample sets the output equal to the input
      y0[0] = u0[0];
 For complex signals use: y0[0].re = u0[0].re;
      y0[0].im = u0[0].im;
      y1[0].re = u1[0].re;
      y1[0].im = u1[0].im;
 */
y0[0]=pid( u0[0] , (double*)P , (double*)I, (double*)D);
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_END --- EDIT HERE TO _BEGIN */
}
```

Header files which includes custom data type definitions.
General purpose libraries
Width of on input and output functions
Output function.
Custom Function code

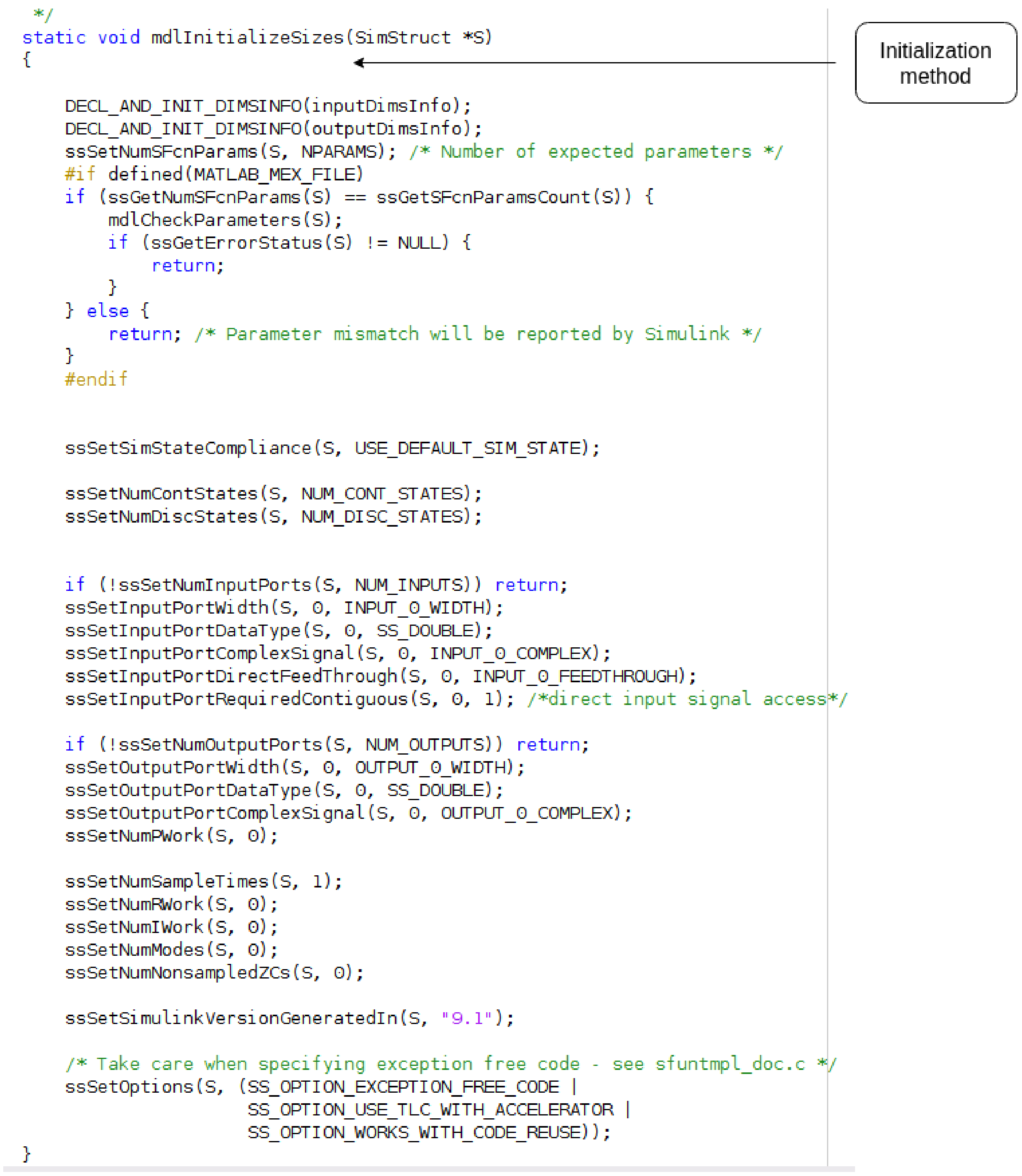
C file



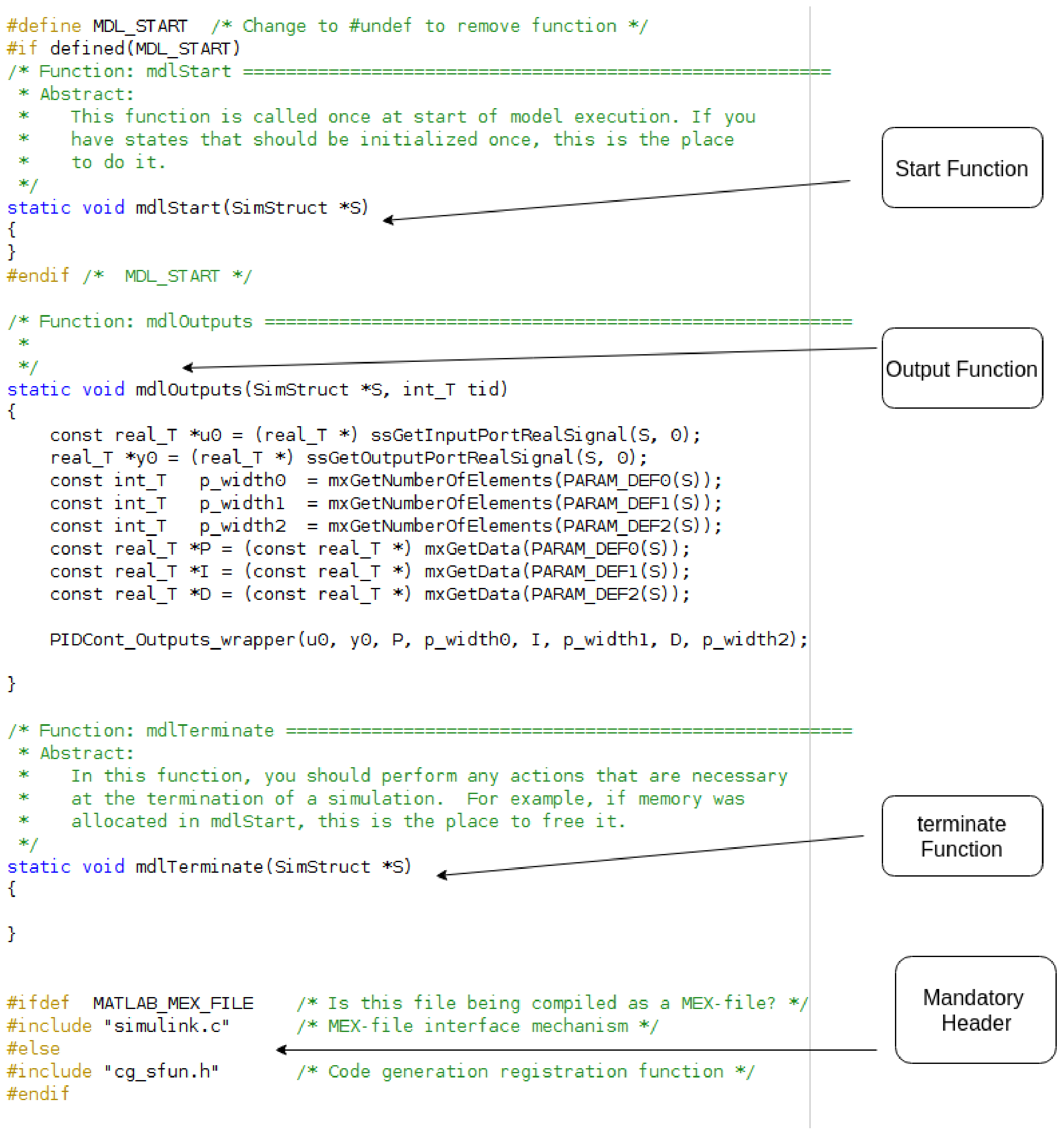
```
#define S_FUNCTION_LEVEL 2
#define S_FUNCTION_NAME PIDCont
/*<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<*/
/* %%%-SFUNWIZ_defines_Changes_BEGIN --- EDIT HERE TO _END */
#define NUM_INPUTS            1
/* Input Port  0 */
#define IN_PORT_0_NAME        u0
#define INPUT_0_WIDTH         1
#define INPUT_DIMS_0_COL      1
#define INPUT_0_DTYPE         real_T
#define INPUT_0_COMPLEX       COMPLEX_NO
#define IN_0_FRAME_BASED      FRAME_NO
#define IN_0_BUS_BASED        0
#define IN_0_BUS_NAME
#define IN_0_DIMS             1-D
#define INPUT_0_FEEDTHROUGH   1
#define IN_0_ISSIGNED         0
#define IN_0_WORDLENGTH       8
#define IN_0_FIXPOINTSCALING  1
#define IN_0_FRACTIONLENGTH   9
#define IN_0_BIAS             0
#define IN_0_SLOPE            0.125

#define NUM_OUTPUTS           1
/* Output Port  0 */
#define OUT_PORT_0_NAME       y0
#define OUTPUT_0_WIDTH        1
#define OUTPUT_DIMS_0_COL     1
#define OUTPUT_0_DTYPE        real_T
#define OUTPUT_0_COMPLEX      COMPLEX_NO
#define OUT_0_FRAME_BASED     FRAME_NO
#define OUT_0_BUS_BASED       0
#define OUT_0_BUS_NAME
#define OUT_0_DIMS            1-D
#define OUT_0_ISSIGNED        1
#define OUT_0_WORDLENGTH      8
#define OUT_0_FIXPOINTSCALING 1
#define OUT_0_FRACTIONLENGTH  3
#define OUT_0_BIAS            0
#define OUT_0_SLOPE           0.125

#define NPARAMS               3
/* Parameter 0 */
#define PARAMETER_0_NAME      P
#define PARAMETER_0_DTYPE     real_T
#define PARAMETER_0_COMPLEX   COMPLEX_NO
/* Parameter 1 */
#define PARAMETER_1_NAME      I
#define PARAMETER_1_DTYPE     real_T
#define PARAMETER_1_COMPLEX   COMPLEX_NO
```


Simstruc to declare the properties
S function methods to set the simulation conditions

```
/*=====================================================================*/
#include "simstruc.h"
#define PARAM_DEF0(S) ssGetSFcnParam(S, 0)
#define PARAM_DEF1(S) ssGetSFcnParam(S, 1)
#define PARAM_DEF2(S) ssGetSFcnParam(S, 2)

#define IS_PARAM_DOUBLE(pVal) (mxIsNumeric(pVal) && !mxIsLogical(pVal) &&\
!mxIsEmpty(pVal) && !mxIsSparse(pVal) && !mxIsComplex(pVal) && mxIsDouble(pVal))

extern void PIDCont_Outputs_wrapper(const real_T *u0,
            real_T *y0,
            const real_T *P, const int_T p_width0,
            const real_T *I, const int_T p_width1,
            const real_T *D, const int_T p_width2);
/*====================*
 * S-function methods *
 *====================*/
#define MDL_CHECK_PARAMETERS
#if defined(MDL_CHECK_PARAMETERS) && defined(MATLAB_MEX_FILE)
/* Function: mdlCheckParameters =============================================
 * Abstract:
 *    Verify parameter definitions and types.
 */
static void mdlCheckParameters(SimStruct *S)
{
    int paramIndex  = 0;
    bool invalidParam = false;
    /* All parameters must match the S-function Builder Dialog */

    {
        const mxArray *pVal0 = ssGetSFcnParam(S, 0);
        if (!IS_PARAM_DOUBLE(pVal0)) {
            invalidParam = true;
            paramIndex = 0;
            goto EXIT_POINT;
        }
    }

    {
        const mxArray *pVal1 = ssGetSFcnParam(S, 1);
        if (!IS_PARAM_DOUBLE(pVal1)) {
            invalidParam = true;
            paramIndex = 1;
            goto EXIT_POINT;
        }
    }

    {
        const mxArray *pVal2 = ssGetSFcnParam(S, 2);
        if (!IS_PARAM_DOUBLE(pVal2)) {
            invalidParam = true;
```



```
 */
static void mdlInitializeSizes(SimStruct *S)
{

    DECL_AND_INIT_DIMSINFO(inputDimsInfo);
    DECL_AND_INIT_DIMSINFO(outputDimsInfo);
    ssSetNumSFcnParams(S, NPARAMS); /* Number of expected parameters */
    #if defined(MATLAB_MEX_FILE)
    if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {
        mdlCheckParameters(S);
        if (ssGetErrorStatus(S) != NULL) {
            return;
        }
    } else {
        return; /* Parameter mismatch will be reported by Simulink */
    }
    #endif


    ssSetSimStateCompliance(S, USE_DEFAULT_SIM_STATE);

    ssSetNumContStates(S, NUM_CONT_STATES);
    ssSetNumDiscStates(S, NUM_DISC_STATES);


    if (!ssSetNumInputPorts(S, NUM_INPUTS)) return;
    ssSetInputPortWidth(S, 0, INPUT_0_WIDTH);
    ssSetInputPortDataType(S, 0, SS_DOUBLE);
    ssSetInputPortComplexSignal(S, 0, INPUT_0_COMPLEX);
    ssSetInputPortDirectFeedThrough(S, 0, INPUT_0_FEEDTHROUGH);
    ssSetInputPortRequiredContiguous(S, 0, 1); /*direct input signal access*/

    if (!ssSetNumOutputPorts(S, NUM_OUTPUTS)) return;
    ssSetOutputPortWidth(S, 0, OUTPUT_0_WIDTH);
    ssSetOutputPortDataType(S, 0, SS_DOUBLE);
    ssSetOutputPortComplexSignal(S, 0, OUTPUT_0_COMPLEX);
    ssSetNumPWork(S, 0);

    ssSetNumSampleTimes(S, 1);
    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

    ssSetSimulinkVersionGeneratedIn(S, "9.1");

    /* Take care when specifying exception free code - see sfuntmpl_doc.c */
    ssSetOptions(S, (SS_OPTION_EXCEPTION_FREE_CODE |
                     SS_OPTION_USE_TLC_WITH_ACCELERATOR |
                     SS_OPTION_WORKS_WITH_CODE_REUSE));
}
```



```
#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
/* Function: mdlStart =======================================================
 * Abstract:
 *    This function is called once at start of model execution. If you
 *    have states that should be initialized once, this is the place
 *    to do it.
 */
static void mdlStart(SimStruct *S)
{
}
#endif /*  MDL_START */

/* Function: mdlOutputs =======================================================
 *
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    const real_T *u0 = (real_T *) ssGetInputPortRealSignal(S, 0);
    real_T *y0 = (real_T *) ssGetOutputPortRealSignal(S, 0);
    const int_T   p_width0  = mxGetNumberOfElements(PARAM_DEF0(S));
    const int_T   p_width1  = mxGetNumberOfElements(PARAM_DEF1(S));
    const int_T   p_width2  = mxGetNumberOfElements(PARAM_DEF2(S));
    const real_T *P = (const real_T *) mxGetData(PARAM_DEF0(S));
    const real_T *I = (const real_T *) mxGetData(PARAM_DEF1(S));
    const real_T *D = (const real_T *) mxGetData(PARAM_DEF2(S));

    PIDCont_Outputs_wrapper(u0, y0, P, p_width0, I, p_width1, D, p_width2);

}

/* Function: mdlTerminate =====================================================
 * Abstract:
 *    In this function, you should perform any actions that are necessary
 *    at the termination of a simulation.  For example, if memory was
 *    allocated in mdlStart, this is the place to free it.
 */
static void mdlTerminate(SimStruct *S)
{

}


#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```