

# Quick YALMIP Manual

**Based on:** <http://users.isy.liu.se/johanl/yalmip/pmwiki.php?n=Main.HomePage>

# 1. Commands

# sdpvar

**sdpvar** Create symbolic decision variable

You can create a sdpvar variable by:

<code>X = sdpvar(n)</code>	Symmetric nxn matrix
<code>X = sdpvar(n,n)</code>	Symmetric nxn matrix
<code>X = sdpvar(n,m)</code>	Full nxm matrix (n~m)

Definition of multiple scalars can be simplified

**sdpvar** x y z w

The parametrizations supported are

<code>X = sdpvar(n,n,'full')</code>	Full nxn matrix
<code>X = sdpvar(n,n,'symmetric')</code>	Symmetric nxn matrix
<code>X = sdpvar(n,n,'diagonal')</code>	Diagonal matrix
<code>X = sdpvar(n,n,'toeplitz')</code>	Symmetric Toeplitz
<code>X = sdpvar(n,n,'hankel')</code>	Unsymmetric Hankel (zero below the first anti-diagonal)
<code>X = sdpvar(n,n,'rhankel')</code>	Symmetric Hankel
<code>X = sdpvar(n,n,'skew')</code>	Skew-symmetric
<code>X = sdpvar(n,n,'diagonal')</code>	Diagonal

The letters 'sy','f','ha','t' and 'sk' are searched for in the third argument hence `sdpvar(n,n,'toeplitz')` gives the same result as `sdpvar(n,n,'t')`

Only square Toeplitz and Hankel matrices are supported

A scalar is defined as a 1x1 matrix

Higher-dimensional matrices are also supported, although this currently is an experimental feature with limited use. The type flag applies to the lowest level slice.

`X = sdpvar(n,n,n,'full')` Full nxnxd matrix

In addition to the matrix type, a fourth argument can be used to obtain a complex matrix. All the matrix types above apply to a complex matrix, and in addition a Hermitian type is added

`X = sdpvar(n,n,'hermitian','complex')` Complex Hermitian nxn matrix (`X=X'=conj(X.')`)

The other types are obtained as above

<code>X = sdpvar(n,n,'symmetric','complex')</code>	Complex symmetric nxn matrix ( <code>X=X.')</code>
<code>X = sdpvar(n,n,'full','complex')</code>	Complex full nxn matrix

... and the same for Toeplitz, Hankel and skew-symmetric

## See also

[intvar](#), [binvar](#), [methods\('sdpvar'\)](#), [see](#)

## Overloaded methods:

[ndsdpvar/sdpvar constraint/sdpvar blkvar/sdpvar](#)

# create\_CHS

**create\_CHS** Internal function to define matrices for MPC problem

# optimizer

**optimizer** Container for optimization problem

`OPT = optimizer(Constraints,Objective,options,x,u)` exports an object that contains precompiled numerical data to be solved for varying arguments `x`, returning the optimal value of the expression `u`.

**optimizer** typically only works efficiently if the varying data `x` enters the optimization problem affinely. If not, the precompiled problems will be nonconvex, despite the problem being simple for a fixed value of the parameter (see Wiki for beta support of a much more general optimizer)

In principle, if an optimization problem has a parameter `x`, and we repeatedly want to solve the problem for varying `x` to compute a variable `u`, we can, instead of repeatedly constructing optimization problems for fixed values of `x`, introduce a symbolic `x`, and then simply add an equality

```
solvesdp([Constraints,x == value],Objective);
uopt = double(u)
```

There will still be overhead from the `SOLVESDP` call, so we can precompile the whole structure, and let `YALMIP` handle the addition of the equality constraint for the fixed value, and automatically extract the solution variables we are interested in

```
OPT = optimizer(Constraints,Objective,options,x,u)
uopt1 = OPT{value1}
uopt2 = OPT{value2}
uopt3 = OPT{value3}
```

By default, display is turned off (since `optimizer` is used in situations where many problems are solved repeatedly. To turn on display, set the `verbose` option in `sdpssetting` to 2.

## Example

The following problem creates an LP with varying upper and lower bounds on the decision variable.

The optimizing argument is obtained by indexing (with `{}`) the optimizer object with the point of interest. The argument should be a column vector (if the argument has a width larger than 1, `YALMIP` assumes that the optimal solution should be computed in several points)

```
A = randn(10,3);
b = rand(10,1)*19;
c = randn(3,1);

z = sdpvar(3,1);
sdpvar UB LB

Constraints = [A*z <= b, LB <= z <= UB];
Objective = c'*z
% We want the optimal z as a function of [LB;UB]
optZ = optimizer(Constraints,Objective,[],[LB; UB],z);
```

```
% Compute the optimal z when LB=1, UB = 3;
zopt = optZ{[1; 3]}

% Compute two solutions, one for (LB,UB) [1;3] and one for (LB,UB) [2;6]
zopt = optZ{[1; 3], [2;6]}

% A second output argument can be used to catch infeasibility
[zopt,infeasible] = optZ{[1; 3]}

% To avoid the need to vectorize in order to handle multiple
  parameters, a cell-based format can be used, both for inputs and
  outputs. Note that the optimizer object now is called with a cell
  and returns a cell

optZ = optimizer(Constraints,Objective,[],{LB,UB},{z,sum(z)})
[zopt,infeasible] = optZ{{1,3}};
zopt{1}
zopt{2}
```

## Overloaded methods:

[optproblem/optimizer](#)

# robustify

**robustify** Derives robust counterpart.

`[Frobust,objrobust,failure] = robustify(F,h,options)` is used to derive the robust counterpart of an uncertain YALMIP model.

$$\begin{array}{ll} \min & h(x,w) \\ \text{subject to} & \\ & F(x,w) \succeq 0 \quad \text{for all } w \text{ in } W \end{array}$$

The constraints and objective have to satisfy a number of conditions for the robustification to be possible. Please refer to the YALMIP Wiki for the current assumptions.

Some options for the robustification strategies can be altered via the solver tag 'robust' in `sdpssettings`

- 'robust.lplp' : Controls how linear constraints with affine parameterization in an uncertainty with polytopic description is handled. Can be either 'duality' or 'enumeration'
- 'robust.auxred': Controls how uncertainty dependent auxiliary variables are handled  
Can be either 'projection' or 'enumeration' (exact), or 'none' or 'affine' (conservative)
- 'robust.reducedual' Controls if the system equality constraints derived when using the duality filter should be eliminated, thus reducing the number of variables, possibly destroying sparsity .
- 'robust.polya' : Controls the relaxation order of polynomials. If set to NAN, the polynomials will be eliminated by forcing the coefficients to zero

## See also

[uncertain](#)

## Overloaded methods:

[optproblem/robustify](#)

# YALMIP Wiki

## Solvesdp

[solvesdp](#) is the common function for solving standard optimization problems.

### Syntax

```
diagnostics = solvesdp(Constraints, Objective, options)
```

### Examples

A linear program  $\{\min c^T x \text{ subject to } Ax \leq b\}$  can be solved with the following piece of code

```
x = sdpvar(length(c), 1);
F = [A*x<=b];
h = c'*x;
solvesdp(F, h);
solution = double(x);
```

If we only look for a feasible solution, we can omit the objective function

```
solvesdp(F);
```

A diagnostic structure is returned from solvesdp, which can be used, e.g, to check feasibility (see [yalmiperror](#) for the error codes)

```
diagnostics = solvesdp(F);
if diagnostics.problem == 0
    disp('Feasible')
elseif diagnostics.problem == 1
    disp('Infeasible')
else
    disp('Something else happened')
end
```

Solving the feasibility problem with a particular solver, e.g. [QUADPROG](#), can be done with

```
solvesdp(F, [], sdpsettings('solver', 'quadprog'));
```

Minimization is assumed, hence if we want to maximize, we simply flip the sign of the objective.

```
solvesdp(F, -h);
```

For more examples, check out the [Examples](#) and [Tutorials](#).

### Related commands

[sdpvar](#), [set](#), [sdpsettings](#), [solvesos](#), [solvemoment](#), [solvemp](#)

Most common  
[sdpvar](#)

[sdpsettings](#)

[solvesdp](#)

Variable declaration  
[binvar](#)

[blkvar](#)

[intvar](#)

[sdpvar](#)

[semivar](#)

[uncertain](#)

Variable manipulation  
[assign](#)

[coefficients](#)

[degree](#)

[dissect](#)

[double](#)

[hessian](#)

[is](#)

[jacobian](#)

[kyp](#)

[linearize](#)

[lowrank](#)

[monolist](#)

[plot](#)

[polynomial](#)

[sdisplay](#)

[sparse](#)

[unblkdiag](#)

Operators

[abs](#)

[entropy](#)

[geomean](#)

[huber](#)

[iff](#)

[implies](#)

[logdet](#)

[logistic](#)

[logsumexp](#)

[median](#)

[nnz](#)



# YALMIP Wiki

## Solvemp

[solvemp](#) is used for solving multiparametric problems.

### Syntax

```
[mptsol,diagnostic,u,J,U] = solvemp(F,h,ops,x)
```

### Examples

[solvemp](#) is used in the same way as [solvesdp](#), the only difference being the fourth additional argument to define the parametric variables. See the numerous [multiparametric examples](#) for details.

### See also

[sdpvar](#), [sdpsettings](#), [solvesdp](#)

Most common

[sdpvar](#)

[sdpsettings](#)

[solvesdp](#)

Variable declaration

[binvar](#)

[blkvar](#)

[intvar](#)

[sdpvar](#)

[semivar](#)

[uncertain](#)

Variable manipulation

[assign](#)

[coefficients](#)

[degree](#)

[dissect](#)

[double](#)

[hessian](#)

[is](#)

[jacobian](#)

[kyp](#)

[linearize](#)

[lowrank](#)

[monolist](#)

[plot](#)

[polynomial](#)

[sdisplay](#)

[sparse](#)

[unblkdiag](#)

Operators

[abs](#)

[entropy](#)

[geomean](#)

[huber](#)

[iff](#)

[implies](#)

[logdet](#)

[logistic](#)

[logsumexp](#)

[median](#)

[nnz](#)

## 2. Wiki

# YALMIP Wiki

## Basics

The following piece of code introduces essentially everything you ever need to learn. It defines variables, constraints, objectives, options, checks result and extracts solution (Note that the code specifies the solver to [CPLEX](#). If you don't have [CPLEX](#) installed, simply remove that part of the option list if you want to run the code).

```
x = sdpvar(2,1);
Constraints = [sum(x) <= 1, x(1)==0, x(2) >= 0.5];
Objective = x'*x;
options = sdpsettings('verbose',1,'solver','cplex','cplex.qpmethod',1);
sol = solvesdp(Constraints,Objective,options);
if sol.problem == 0
    solution = double(x);
else
    display('Hmm, something went wrong!');
    sol.info
    yalmiperror(sol.problem)
end
```

Having seen that, let us start from the beginning.

## YALMIPs symbolic variable

The most important command in YALMIP is [sdpvar](#). This command is used to define decision variables. To define a matrix (or scalar) P with n rows and m columns, we write

```
P = sdpvar(n,m)
```

**A square matrix is symmetric by default!** To obtain a fully parameterized (i.e. not necessarily symmetric) square matrix, a third argument is needed.

```
P = sdpvar(3,3,'full')
```

The third argument can be used to obtain a number of pre-defined types of variables, such as Toeplitz, Hankel, diagonal, symmetric and skew-symmetric matrices. See the help text on [sdpvar](#) for details. Alternatively, the associated MATLAB commands can be applied to a vector.

```
x = sdpvar(n,1);
D = diag(x); % Diagonal matrix
H = hankel(x); % Hankel matrix
T = toeplitz(x); % Hankel matrix
```

Scalars can be defined in three different ways.

```
x = sdpvar(1,1); y = sdpvar(1,1);
x = sdpvar(1); y = sdpvar(1);
sdpvar x y
```

Note that due to a bug in MATLAB, the last command-line syntax fails in some cases (inside functions), if the variable name is the same as some built-in function or variable (i, j, e, beta, gamma).

The [sdpvar](#) objects are manipulated in MATLAB as any other variable and most functions are overloaded. Hence, the following commands are valid

```
P = sdpvar(3,3) + diag(sdpvar(3,1));
X = [P P; eye(length(P))] + 2*trace(P);
Y = X + sum(sum(P*rand(length(P)))) + P(end,end)+hankel(X(:,1));
```

In some situations, coding is simplified with a multi-dimensional variable. This is supported in YALMIP with two different constructs, cell arrays and multi-dimensional [sdpvar](#) objects.

The cell array is nothing but an abstraction of the following code

```
for i = 1:5
    X{i} = sdpvar(2,3);
```

[Introduction](#)

[Installation](#)

[Basics \(start here!\)](#)

[Standard problems](#)

[Linear programming](#)

[Quadratic programming](#)

[Second order cone programming](#)

[Semidefinite programming](#)

[Determinant maximization](#)

[Geometric programming](#)

[General convex programming](#)

[Advanced topics](#)

[Nonlinear operators](#)

[Robust optimization](#)

[Automatic dualization](#)

[Multiparametric programming](#)

[Bilevel programming](#)

[Sum-of-squares](#)

[Moment relaxations](#)

[Integer programming](#)

[Global optimization](#)

[Logic programming](#)

[Big-M and convex hulls](#)

[KYP problems](#)

[Rank problems](#)

[Auxillary](#)

[Complex problems](#)

[Duality](#)

[Inside YALMIP](#)

```
end
```

By using vector dimensions in `sdpvar`, the same cell array can be setup as follows

```
X = sdpvar([2 2 2 2 2],[3 3 3 3 3]);
```

The cell array can now be used as usual in MATLAB.

The drawback with the approach above is that the variable `X` not can be used directly, as a standard `sdpvar` object. As an alternative, a completely general multi-dimensional `sdpvar` is available. We can create an essentially equivalent object with this call.

```
X = sdpvar(2,3,5);
```

The difference is that we can operate directly on this object, using standard MATLAB code.

```
Y = sum(X,3)
X(:, :, 2)
```

Note that the two first slices are symmetric (if the two first dimensions are the same), according to standard YALMIP syntax. To create a fully parameterized higher-dimensional, use trailing flags as in the standard case.

```
X = sdpvar(2,2,2,2,'full');
```

For an illustration of multi/dimensional variables, check out the [Sudoku example](#).

## Constraints

To define a collection of constraints, we simply define and concatenate them. The meaning of a constraint is context-dependent. If the left-hand side and right-hand side are Hermitian, the constraint is interpreted in terms of positive definiteness, otherwise element-wise. Hence, declaring a symmetric matrix and a positive definiteness constraint is done with

```
n = 3;
P = sdpvar(n,n);
C = [P>=0];
```

while a symmetric matrix with positive elements is defined with, e.g.,

```
P = sdpvar(n,n);
C = [P(:)>=0];
```

Note that this defines the off-diagonal constraints twice. A good SDP solver will perhaps detect this during preprocessing and reduce the model, but we can of-course define only the unique elements manually using standard MATLAB code

```
C = [triu(P)>=0];
```

or

```
C = [P(find(triu(ones(n))))>=0];
```

According to the rules above, a non-square matrix (or generally a non-symmetric) with positive elements can be defined using the `>=` operator immediately

```
P = sdpvar(n,2*n);
C = [P>=0];
```

and so can a fully parameterized matrix with positive elements

```
P = sdpvar(n,n,'full');
C = [P>=0];
```

A list of several constraints is defined by just adding or, alternatively, concatenating them.

```
P = sdpvar(n,n);
C = [P>=0] + [P(1,1)>=2];
C = [P>=0, P(1,1)>=2];
```

Of course, the involved expressions can be arbitrary `sdpvar` objects, and equality constraints (`==`) can be defined, as well as constraints using `<=`.

```
C = [ P >= 0, P(1,1) <= 2, sum(sum(P)) == 10 ];
```

A convenient way to define several constraints is to use double-sided constraints.

```
F = [ 0 <= P(1,1) <= 2 ];
```

After having defined variables and constraints, you are ready to solve problems. Check out the remaining tutorials to learn this.

## YALMIP Wiki

## Semidefinite Programming

 Semidefinite programming  [sdptutorial.m](#)

This example illustrates the definition and solution of a simple semidefinite programming problem.

Given a linear dynamic system  $\dot{x} = Ax$ , our goal is to prove stability by finding a symmetric matrix  $P$  satisfying

$$\begin{aligned} A^T P + P A &\preceq 0 \\ P &\succeq 0 \end{aligned}$$

Define a stable matrix  $A$  and symmetric matrix  $P$  (remember: square matrices are symmetric by default)

```
A = [-1 2 0;-3 -4 1;0 0 -2];
P = sdpvar(3,3);
```

Having defined  $P$ , we are ready to define the semidefinite constraints.

```
F = [P >= 0, A'*P+P*A <= 0];
```

To avoid the zero solution on this homogeneous problem, we constrain the trace of the matrix (Of course, this is not the only way. We could have used, e.g., the dehomogenizing constraint  $P \succeq I$  instead)

```
F = [F, trace(P) == 1];
```

At this point, we are ready to solve our problem. But first, we display the collection of constraints to see what we have defined.

```
F
+++++
| ID|      Constraint|      Type|
+++++
| #1| Numeric value| Matrix inequality 3x3|
| #2| Numeric value| Matrix inequality 3x3|
| #3| Numeric value| Equality constraint 1x1|
+++++
```

We only need a feasible solution, so one argument is sufficient when we call `solvesdp` to solve the problem.

```
solvesdp(F);
Pfeasible = double(P);
```

The resulting constraint satisfaction is easily investigated with `checkset`.

```
checkset(F)
+++++
| ID|      Constraint|      Type| Primal residual| Dual residual|
+++++
| #1| Numeric value| Matrix inequality|      0.20138|      8.2785e-016|
| #2| Numeric value| Matrix inequality|      1.1397|      3.6687e-016|
| #3| Numeric value| Equality constraint|     -2.276e-015|     -8.1801e-016|
+++++
```

Minimizing, e.g., the top-left element of  $P$  is done by specifying an objective function.

```
F = [P >= 0, A'*P+P*A <= 0, trace(P)==1];
solvesdp(F,P(1,1));
```

We can easily add additional linear inequality constraints. If we want to add the constraint that all off-

[Introduction](#)
[Installation](#)
[Basics \(start here!\)](#)
[Standard problems](#)
[Linear programming](#)
[Quadratic programming](#)
[Second order cone programming](#)
[Semidefinite programming](#)
[Determinant maximization](#)
[Geometric programming](#)
[General convex programming](#)
[Advanced topics](#)
[Nonlinear operators](#)
[Robust optimization](#)
[Automatic dualization](#)
[Multiparametric programming](#)
[Bilevel programming](#)
[Sum-of-squares](#)
[Moment relaxations](#)
[Integer programming](#)
[Global optimization](#)
[Logic programming](#)
[Big-M and convex hulls](#)
[KYP problems](#)
[Rank problems](#)
[Auxillary](#)
[Complex problems](#)
[Duality](#)
[Inside YALMIP](#)


diagonal elements are larger than zero, one approach is (remember, standard MATLAB indexing applies)

```
F = [P >= 0, A'*P+P*A <= 0, trace(P)==1, P([2 3 6])>=0];  
solvesdp(F,P(1,1));
```

Since the variable  $P([2\ 3\ 6])$  is a vector, the constraint is interpreted as a standard linear inequality, according to the rules introduced in the [basic tutorial](#).

## YALMIP Wiki

## Multiparametric Programming

 Multiparametric programming , Model predictive control , Quadratic programming

This tutorial requires the [Multi-Parametric Toolbox \(MPT\)](#).

YALMIP can be used to calculate explicit solutions of linear and quadratic programs by interfacing the [Multi-Parametric Toolbox \(MPT\)](#). This tutorial assumes that the reader is familiar with parametric programming and the basics of [MPT](#).

## Generic example.

Consider the following simple quadratic program in the decision variable  $z$ , solved for a particular value on a parameter  $x$ .

```
A = randn(15,3);
b = rand(15,1);
E = randn(15,2);

z = sdpvar(3,1);
x = [0.1;0.2];

F = [A*z <= b+E*x];
obj = (z-1)'*(z-1);

sol = solvesdp(F,obj);
double(z)
ans =
-0.1454
-0.1789
-0.0388
```

To obtain the parametric solution with respect to  $x$ , we call the function [solvemp](#), and tell the solver that  $x$  is a parametric variable. Moreover, we must add constraints on  $x$  to define the region where we want to compute the parametric solution, the so called exploration set.

```
x = sdpvar(2,1);
F = [A*z <= b+E*x, -1 <= x <= 1];
sol = solvemp(F,obj,[ ],x);
```

The first output is an [MPT](#) structure. In accordance with [MPT](#) syntax, the optimizer for the parametric value (0.1,0.2) is given by the following code.

```
xx = [0.1;0.2];
[i,j] = isinside(sol{1}.Pn,xx)
sol{1}.Fi{j}*xx + sol{1}.Gi{j}
ans =
-0.1454
-0.1789
-0.0388
```

By using more outputs from [solvemp](#), it is possible to simplify things considerably.

```
[sol,diagnostics,aux,Valuefunction,Optimizer] = solvemp(F,obj,[ ],x);
```

The function now returns solutions using YALMIPs [nonlinear operator framework](#). To retrieve the numerical solution for a particular parameter value, simply use [assign](#) and `double` in standard fashion.

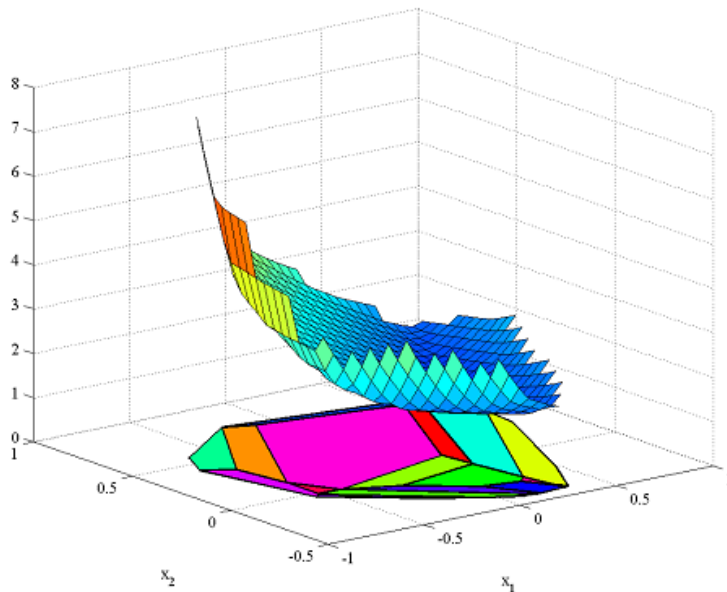
```
assign(x,[0.1;0.2]);
double(Optimizer)
```

Some of the plotting capabilities of [MPT](#) are overloaded for the piecewise functions. Hence, we can plot the piecewise quadratic value function

```
plot(Valuefunction);
figure
plot(Optimizer);
```

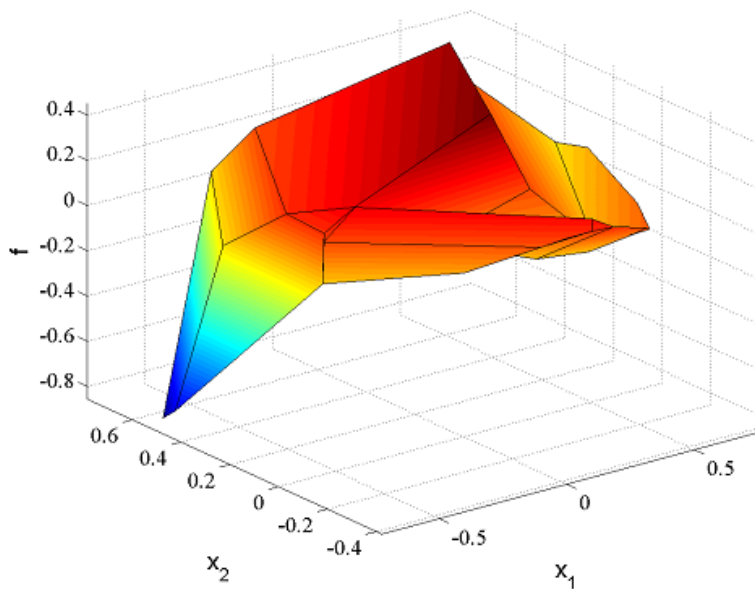
[Introduction](#)
[Installation](#)
[Basics \(start here!\)](#)
[Standard problems](#)
[Linear programming](#)
[Quadratic programming](#)
[Second order cone programming](#)
[Semidefinite programming](#)
[Determinant maximization](#)
[Geometric programming](#)
[General convex programming](#)
[Advanced topics](#)
[Nonlinear operators](#)
[Robust optimization](#)
[Automatic dualization](#)
[Multiparametric programming](#)
[Bilevel programming](#)
[Sum-of-squares](#)
[Moment relaxations](#)
[Integer programming](#)
[Global optimization](#)
[Logic programming](#)
[Big-M and convex hulls](#)
[KYP problems](#)
[Rank problems](#)
[Auxillary](#)
[Complex problems](#)
[Duality](#)
[Inside YALMIP](#)





and plot the piecewise affine optimizer

```
figure
plot(Optimizer(1));
```



## Simple MPC example

Define numerical data for a linear system, prediction matrices, and variables for current state  $x$  and the future control sequence  $U$ , for an MPC problem with horizon 5 (create\_CHS is a function that creates the numerical matrices to describe the linear relation between current state  $x$  and future input sequence  $U$ , to the predicted outputs)

```
N = 5;
A = [2 -1; 1 0];
B = [1; 0];
C = [0.5 0.5];
[H, S] = create_CHS(A, B, C, N);
x = sdpvar(2, 1);
U = sdpvar(N, 1);
```

The future output predictions are linear in the current state and the control sequence.

```
Y = H*x + S*U;
```

We wish to minimize a quadratic cost, compromising between small input and outputs.

```
objective = Y'*Y+U'*U;
```

The input variable has a hard constraint, and so does the output at the terminal state.

```
F = [1 >= U >= -1];
F = [F, 1 >= Y(N) >= -1];
```

We seek the explicit solution  $U(x)$  over the exploration set  $|x| < 5$ .

```
F = [F, 5 >= x >= -5];
```

The explicit solution  $U(x)$  is obtained by calling `solvemp` with the parametric variable  $x$  as the fourth argument. Additionally, since we only are interested in the first element of the solution  $U$ , we use a fifth input to communicate this.

```
[sol,diagnostics,aux,Valuefunction,Optimizer] = solvemp(F,objective,
[],x,U(1));
```

We can plot the overloaded solutions directly

```
figure
plot(Valuefunction)
figure
plot(Optimizer)
```

## Mixed integer multiparametric programming

YALMIP extends the multiparametric solvers in `MPT` by adding support for binary variables in the parametric problems.

Let us solve an extension of the MPC problem from the previous section. To begin with, we formulate a similar problem (shorter horizon and linear cost)

```
N = 3;
A = [2 -1; 1 0];
B = [1; 0];
C = [0.5 0.5];
[H,S] = create_CHS(A,B,C,N);
x = sdpvar(2,1);
U = sdpvar(N,1);
Y = H*x+S*U;

objective = norm(Y,1) + norm(U,1);

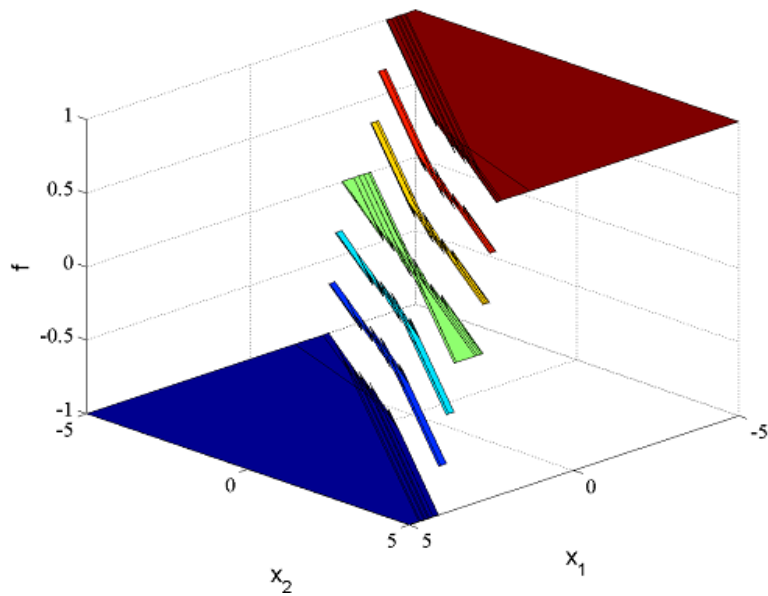
F = [1 >= U >= -1];
F = [F, 5 >= x >= -5];
```

We will now solve this problem under the additional constraints that the input is quantized in steps of  $1/3$ . This can easily be modelled in YALMIP using `ismember`. Note that this nonconvex operator introduce a lot of binary variables, and the MPC problem is most likely solved more efficiently using a [dynamic programming approach](#).

```
F = [F, ismember(U, [-1:1/3:1])];
```

Same commands as before to solve the problem and plot the optimal solution  $U(1)$


```
[sol,diagnostics,aux,Valuefunction,Optimizer] = solvemp(F,objective,
[],x,U(1));
plot(Optimizer);
```



For more examples, see the [dynamic programming example](#), the [robust MPC example](#), the [portfolio example](#), and the [MAXPLUS control example](#).

# YALMIP Wiki

## Robust MPC

 Model predictive control , Robust optimization , Linear programming

You can [download the code here](#).

This example illustrates an application of the [robust optimization framework](#).

Robust optimization can be used for robust control, i.e., derivation of control laws such that constraints are satisfied despite uncertainties in the system, and/or worst-case performance objectives. Various formulations for robust MPC was introduced in ([Löfberg:2003](#)), and we will use YALMIPs [robust optimization feature](#) to derive some of the robustified optimization problems automatically. The example will essentially solve various versions of Example (7.4) in ([Löfberg:2003](#)).

The system description is  $x_{k+1} = Ax_k + Bu_k + Ew_k$ ,  $y_k = Cx_k$

```
A = [2.938 -0.7345 0.25; 4 0 0; 0 1 0];
B = [0.25; 0; 0];
C = [-0.2072 0.04141 0.07256];
E = [0.0625; 0; 0];
```

### Open-loop minimax solution

A prediction horizon  $N=10$  is used. We first derive a symbolic expression of the future outputs by symbolically simulating the predictions. Note that we use an explicit representation of the predictions. In the [dynamic programming](#) example, we used an implicit representation by declaring the state updates via equality constraints. This does not make sense here, since we want the model to hold robustly. An uncertain equality constraint does not make sense.

```
N = 10;
U = sdpvar(N,1);
W = sdpvar(N,1);
x = sdpvar(3,1);

Y = [];
xk = x;
for k = 1:N
    xk = A*xk + B*U(k) + E*W(k);
    Y = [Y; C*xk];
end
```

An alternative could be to use the predefined prediction matrices that are used in the simple MPC example in the [multiparametric tutorial](#).

```
[H,Su] = create_CHS(A,B,C,N);
[H,Sw] = create_CHS(A,E,C,N);
Y = H*x + Su*U + Sw*W;
```

The input is bounded, and the objective is to stay as close as possible to the level  $y=1$  while guaranteeing that  $y < 1$  is satisfied despite the disturbances  $w$ .

```
F = [Y <= 1, -1 <= U <= 1];
objective = norm(Y-1,1) + norm(U,1)*0.01;
```

The uncertainty is known to be bounded.

```
G = [-1 <= W <= 1]
```

To speed up the simulations (the code is still pretty slow), the robustified model is derived once without solving it using [robustify](#)

```
[Frobust,h] = robustify(F + G,objective,[],W);
```

The derived model can now be used a standard YALMIP model, allowing us to simulate the closed-loop system by repeatedly solving the problem.

[Stock portfolio selection](#)

[Sudoku solver](#)

[Experiment design for SYSID](#)

[Unit commitment](#)

[Bilevel programming](#)

[Polytope geometry](#)

[More sum-of-squares](#)

[LTI decay-rate](#)

[LPV control](#)

[Standard MPC](#)

[Hybrid MPC](#)

[Explicit MPC](#)

[Explicit LPV-MPC](#)

[Explicit LPVA-MPC](#)

[Robust MPC](#)

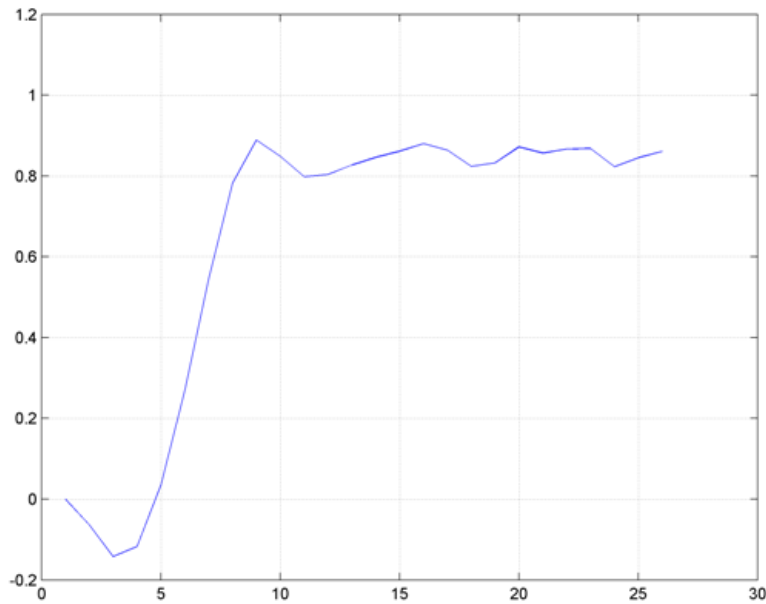
[MAXPLUS MPC](#)

[edit SideBar](#)

```

xk = [0;0;0];
ops = sdpsettings;
for i = 1:25
    solvesdp([Frobust, x == xk(:,end)],h,ops);
    xk = [xk A*xk(:,end) + B*double(U(1)) + E*(-1+2*rand(1))];
end
plot(C*xk)

```



To speed up the simulation further, we can use the `optimizer` construction. It is applicable here since the changing part of the optimization problem is the current state, which enters affinely in the model.

```

controller = optimizer([F, G, uncertain(W)],objective,ops,x,U(1));
xk = [0;0;0];
for i = 1:25
    xk = [xk A*xk(:,end) + B*controller{xk(:,end)} + E*(-1+2*rand(1))];
end
plot(C*xk)

```

Indeed, the solution satisfies the hard constraint, but the steady-state level on  $y$  is far away from the desired level. The reason is the open-loop assumption in the problem. The input sequence computed at time  $k$  has to take all future disturbances into account, and does not use the fact that MPC is implemented in a receding horizon fashion.

A better solution is a closed-loop assumption that exploits the fact that future inputs can be functions of future states. This gives a lot less conservative solution, but the solution is, if not intractable, very hard. Typical solution require dynamic programming strategies, or brute-force enumeration. A tractable alternative was introduced in (Löfberg:2003).

## Approximate closed-loop minimax solution

The idea in (Löfberg:2003) was to parametrize future inputs as affine functions of past disturbances. This, in contrast to parametrization in past states, lead to convex and tractable problems.

We create a causal feedback  $U = LW + V$  and derive the predicted states.

```

V = sdpvar(N,1);
L = sdpvar(N,N,'full').*(tril(ones(N))-eye(N));

U = L*W + V;

Y = [];
xk = x;
for k = 1:N
    xk = A*xk + B*U(k)+E*W(k);
    Y = [Y;C*xk];
end

```

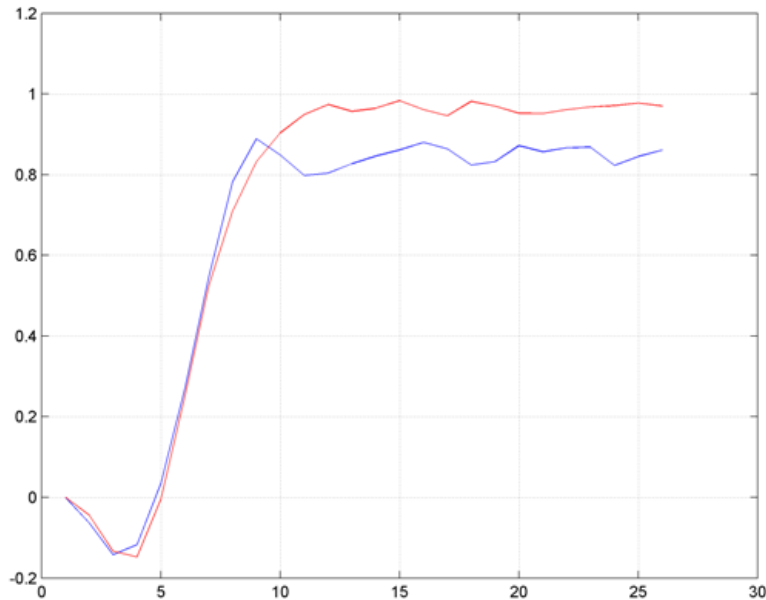
A reasonable implementation of the worst-case scenario with this approximate closed-loop feedback is given by the following code.

```
F = [Y <= 1, -1 <= U <= 1];
objective = norm(Y-1,1) + norm(U,1)*0.01;

[Frobust,h] = robustify([F, G],objective,[],W);

xk = [0;0;0];
ops = sdpsettings;
for i = 1:25
    solvesdp([Frobust, x == xk(:,end)],h,ops);
    xk = [xk A*xk(:,end) + B*double(U(1)) + E*(-1+2*rand(1))];
end

hold on
plot(C*xk,'r')
```



Obviously, the performance is far better (although we admittedly used different disturbance realizations, but the results are consistent if you run the simulations repeatedly).

## Multiparametric solution of approximate closed-loop minimax problem

(This feature is still not working well but require some additional work to improve performance. Coming soon though!)

The robust optimization framework is integrated in the over-all infrastructure in YALMIP. Hence, a model can be robustified, and then sent to the multiparametric solver [MPT](#) in order to get a [multiparametric solution](#).

In our case, we want to have a multi-parametric solution with respect to the state  $x$ . One way to compute the parametric solution is to first derive the robustified model, and send this to the parametric solver.

```
[Frobust,h] = robustify(F + G,objective,[],W);
sol = solvemp(Frobust,h,[],x);
```

Alternatively, we can send the uncertain model directly to [solvemp](#), but we then have declare the uncertain variables via the command [uncertain](#)

```
sol = solvemp([F,G,uncertain(W)],objective,[],x);
```

## Dynamic programming solution to closed-loop minimax problem

It should be mentioned that, for some problems, an exact closed-loop solution can probably be computed more efficiently with dynamic programming along the lines of [the dynamic programming](#)

examples.

Recall the DP code for the [dynamic programming example for LTI systems](#) to solve our problem without any uncertainty.

```
% Model data
A = [2.938 -0.7345 0.25; 4 0 0; 0 1 0];
B = [0.25; 0; 0];
C = [-0.2072 0.04141 0.07256];
E = [0.0625; 0; 0];
nx = 3; % Number of states
nu = 1; % Number of inputs

% Prediction horizon
N = 10;
% States x(k), ..., x(k+N)
x = sdpvar(repmat(nx,1,N), repmat(1,1,N));
% Inputs u(k), ..., u(k+N) (last one not used)
u = sdpvar(repmat(nu,1,N), repmat(1,1,N));

J{N} = 0;
for k = N-1:-1:1
    % Feasible region
    F = [-1 < u{k} < 1, C*x{k} < 1, C*x{k+1} < 1];

    % Bounded exploration space
    % (recommended for numerical reasons)
    F = [F, -100 <= x{k} <= 100];

    % Dynamics
    F = [F, x{k+1} == A*x{k}+B*u{k}];

    % Cost in value iteration
    obj = norm(C*x{k}-1,1) + norm(u{k},1)*0.01 + J{k+1};

    % Solve one-step problem
    [sol{k},diagnost{k},Uz{k},J{k},Optimizer{k}] = solvemp(F,obj,
[],x{k},u{k});
end
```

We will now make some small additions to solve this problem robustly, i.e. minimize worst-case cost and satisfy constraints for all disturbances.

The first change is that we cannot work with equality constraints to define the state dynamics, since the dynamics are uncertain. Instead, we add constraints on the uncertain prediction equations.

Furthermore, the value function  $J\{k+1\}$  is defined in terms of the variable  $x\{k+1\}$ , but since we do not link  $x\{k+1\}$  with  $x\{k\}$  and  $u\{k\}$  with an equality constraint because of the uncertainty, we need to redefine the value function in terms of the uncertain prediction, to make sure that the objective function will be the worst-case cost.

```
% Uncertainty w(k), ..., w(k+N) (last one not used)
w = sdpvar(repmat(1,1,N), repmat(1,1,N));

J{N} = 0;
for k = N-1:-1:1
    % Feasible region
    F = [-1 < u{k} < 1, C*x{k} < 1];
    F = [F, C*(A*x{k} + B*u{k} + E*w{k}) < 1];

    % Bounded exploration space
    F = [F, -100 <= x{k} <= 100];

    % Uncertain value function
    Jw = replace(J{k+1},x{k+1},A*x{k} + B*u{k} + E*w{k});

    % Declare uncertainty model
    F = [F, uncertain(w{k})];
    F = [F, -1 <= w{k} <= 1];

    % Cost in value iteration
    obj = norm(C*x{k}-1,1) + norm(u{k},1)*0.01 + Jw;

    % Solve one-step problem
    [sol{k},diagnost{k},Uz{k},J{k},Optimizer{k}] = solvemp(F,obj,
```

```
[ ], x{k}, u{k});  
end
```

Please note that this multiparametric problem seems to grow large, hence it will take a fair amount of time to compute. Rest assured though, we are constantly working on improving performance in both [MPT](#) and YALMIP.



# YALMIP Wiki

## Nonlinear Operators

### Nonlinear programming

Operators and functions, such as [exp](#) or [abs](#), can be equipped with various properties, which allows YALMIP to, e.g., analyze the optimization problem with respect to convexity, select a suitable way to model the problem, or automatically compute gradients and jacobians. If the problem is convex, YALMIP can sometimes use a [graph representation](#), while a nonconvex problem might require the introduction of [mixed-integer representations](#) or a [callback approach](#).

### Graph-based representations

Graph-based implementations model the operators by using additional variables and constraints, so called [epi](#)- and [hypo-graphs](#). The benefit of this modeling strategy is that YALMIP can derive simple optimization models, such as linear programs, instead of treating the operators as general nonlinearities and use a nonlinear solver.

These graph representations are only valid if the expressions satisfy certain convexity and concavity conditions. Hence, YALMIP has to analyze the problem to ensure that the representation actually can be used.

Graph-based implementations are available for a large number of operators, such as [min](#), [max](#), [abs](#), [sqrt](#), [norm](#) and [geomean](#). Adding new operators is easy, and can be done almost entirely from template code.

More information can be found [here](#).

### Mixed-integer representations

Mixed-integer representations are models that encode an exact representation of an operator by using integer and binary decision variables. The benefit of using such a representation is that the resulting nonconvex problem typically is a well structured problem, such as a mixed-integer linear program. Many of the operators that are implemented using linear programming representable graphs, are also available in a mixed integer representation. If YALMIP fails to propagate convexity, it will switch from graph-based modelling to mixed-integer modelling (unless the option 'allownonconvex' is set to 0). Mixed-integer models are available for, e.g., [min](#), [max](#), [abs](#), and combinatorial and logical operators such as [or<sup>2</sup>](#), [and<sup>2</sup>](#), [min](#), [max](#), [ne<sup>2</sup>](#), [iff](#), [implies](#), [nnz](#), [alldifferent](#), [sort](#) and [ismember](#), and on a higher level, piecewise affine and quadratic functions in connection with [MPT](#).

More information can be found [here](#).

### Evaluation-based representations

A third way to model operators in YALMIP is by using simple callback evaluations. Operators modelled this way can also be equipped with convexity information, and thus fit into YALMIPs convexity propagations. However, since they are based on callbacks, they can only be used together with general purpose optimization solvers, such as [fmincon](#).

Most of MATLABs built-in nonlinear functions, such as [exp](#) and [log](#), are available as evaluation-based representations in YALMIP, and you can easily add your own.

More information can be found [here](#).

[Introduction](#)
[Installation](#)
[Basics \(start here!\)](#)
[Standard problems](#)
[Linear programming](#)
[Quadratic programming](#)
[Second order cone programming](#)
[Semidefinite programming](#)
[Determinant maximization](#)
[Geometric programming](#)
[General convex programming](#)
[Advanced topics](#)
[Nonlinear operators](#)
[Robust optimization](#)
[Automatic dualization](#)
[Multiparametric programming](#)
[Bilevel programming](#)
[Sum-of-squares](#)
[Moment relaxations](#)
[Integer programming](#)
[Global optimization](#)
[Logic programming](#)
[Big-M and convex hulls](#)
[KYP problems](#)
[Rank problems](#)
[Auxillary](#)
[Complex problems](#)
[Duality](#)
[Inside YALMIP](#)