Simplified reactor physics for RBWR style games

Author:

Delfino Delphis

Reviewers:

Stefan (inami_)
Leaf (keshud)

Typesetter:

Stefan (inami_)

Date: 2023-12-24

RBWR is a computer game developed in roblox's engine which tries to simulate operations of a BWR type reactor. The game had quite an impact on the community and many programmers tried to build their own reactor games although they often had problems in implementing physics in their simulation. Based on the experiences of RBWR I present you a short guide on how to start building such a game.

The Philosophy of RBWR was to make things as simple as possible while trying to not sacrificing on general feel of how a nuclear reactor works. Another goal was to have a working product ready as soon as possible and then improving it later rather than building a complete game from the start. I strongly suggest this approach because it's not that difficult to add thousands of switches and real life systems but making it all work is an enormous effort. You can't even imagine how many problems you will face and in how many ways different systems can interact with each other. If you have a simple environment with working autocontrol, you can continue to add one system after another testing how it interacts with all other parts and still keeping the game playable. Also feedback from the community testing a working product will give you a lot of ideas which path to follow ie. what needs more complex simulation and what doesn't need to be simulated at all.

RBWR was built around gameplay rather than a full scale simulation. If you want an experience where reactor startup takes hours like in real life you'd probably need a more complex approach than presented here, although your playerbase will be small. The main assumption behind RBWR was to have a realistic and cooperative experience giving a realistic feel of the operations but still managable and attractive for a casual player. Therefore RBWR rarely uses real physical constants, often relying on some custom made constants which are there to tune the experience as we want it.

For example: With water level at nominal and the reactor at full power, how long would it take for the whole water to boil out from an initial temperature of 100 degrees? Of course if you have all the data like volume of the reactor vessel, amount of water and reactor power, it's not difficult. It's just a matter calculating a simple equation for the specific heat of water and evaporation. But the question at the end will be – does it fit the gameplay I want to design? From my perspective you should take the opposite approach. First answer that question and then tune the physics to achieve that.

If you are designing a more arcade style experience where a lot is happening, probably something like 1 minute would be a good bet. If you lose feedwater pumps at full power it's a matter of seconds before something bad happens. If you want a more realistic approach you could pick say 6 minutes. In that case there is some time to react. Maybe even longer time? Now you have to realize that if it takes 6 minutes to boil out all water at 100% of reactor power, it will take around an hour to do it at 10% power. Some users might think that the experience is boring if it takes so much time for something bad to happen at powers where you usually runup your turbine. And the decay heat which is usually 1-3% of reactor power, would take hours to have any impact and at this moment you could even consider not simulating it at all.

Therefore I suggest you to use this fine tuning principle although if you want a more realistic simulation with use of real physical constants, that's also fine, you can still do it in most cases but you will need to research for some more real life data to achieve it.

Reactor

The Reactor is the main component that drives your simulation. While nuclear physics is complicated it is really easy to simulate behavior of a real reactor with some simplifications. It is important to make it right here because how the reactor works will impact your whole experience a lot. I've seen many experiences where a lot of effort was put to build a nice looking control room, but without proper simulation of the reactor itself they just don't feel right.

Nuclear Reactors physics essentially comes down to an exponential function, that's basically all you need to know. The whole process is complicated, with prompt; delayed; fast and thermal neutrons, but you don't need to simulate those at all as every reactor will feel very simillar at the end with a simple exponential behavior, no matter how complicated the physics are behind it. Exponential function is a function where slope (rate of change) is proportional to the value of the function itself. Therefore if it takes a second for reactor power to raise from 1% to 2% it will take another second to raise to 4% and another one to 8% and so on. As you can see in each timestep power doubles, so power in new timestep is proportional to the power in the old timestep by a factor of 2. This is just natural to the process of fission, as every decay produces some neutrons, which decay more atoms, which produces more neutron and so on. So the current number of neutrons mostly depend on how many neutrons were available in the last step.

Timestep is a very important part of your simulation and you need to consider it very carefully. RBWR uses a 1 second timestep and I have never had any complaints on that but you can easily use 10 steps per second or any other number. The more complex the physics and the more calculations needed the longer your timestep should be, so the experience doesn't become laggy. As most of the physics will use the same timestep it is important to pick a propper value because as everything is tuned to that timestep, changing the timestep will require to rework all constants that you came up with.

In RBWR I use the number of neutrons as a measure of power, because it's easily scalable but you can use just reactor power instead (in percent or fraction of total power). This is how I define power:

```
local NumberOfNeutrons
local MaxNeutrons
local ReactorPower = NumberOfNeutrons / MaxNeutrons
```

This kind of approach allows me to scale the power by changing `MaxNeutrons`. The higher number you put here, the longer it will take for the reactor to reach full power from criticality. Usually reactors become critical at very low powers (the so called intermediate power range), which is a tiny fraction of full reactor power, but you can just tune it to your

startup procedure and to generally make the startup feel like you want it to. Also please note that the number of neutrons has no physical meaning here. It's just a number.

Reactivity formula

Now we come to the basic reactor formula that drives the whole reactor:

```
local NumberOfNeutrons
local MaxNeutrons
local ReactorPower = NumnerOfNeutrons/MaxNeutrons

local NumberOfNeutrons -- Power now

local oldNumberOfNeutrons -- Power in previous timestep

local SomeFactors
local IdleNeutrons

NumberOfNeutrons = SomeFactors * (oldNumberOfNeutrons + IdleNeutrons)
```

Believe it or not, but this one equations is basically everything that drives the reactor model. As mentioned before new power is dependant on the old power by the factor which I called `SomeFactors` here. We will discuss these factors in a moment. `IdleNeutrons` is only there to allow for startup. If we put the value as 0, we can never start because any value times 0 equals 0. So we can never allow power to drop to zero. The value you pick for this constant defines at what numbers you will operate. For RBWR I use:

```
local IdleNeutrons = 1000
local MaxNeutrons = 100000000000 -- 10e11
```

But you can use different numbers to get a different feel.

Absorption and moderation

The number `SomeFactors` defines the state of the reactor. If the number is 1 then the reactor is ideally critical and it will stay at constant power. Higher value will mean exponential raise of power and lower than one an exponential drop. This constant depends on two factors which is absorption and moderation. Absorption mainly comes from control rods and other factors like xenon abundance.

Light water reactors mainly use slow, also called thermal neutrons meanwhile the ones produced in Fission decays are fast. In order to slow down the neutrons we use a moderator which is just water. Other reactors can use different types of moderators for example graphite or in the case of fast reactors none at all because they rely on fast neutrons. The pool of available neutrons for us depends on the amount of moderation. Moderation and absorption are two sides of the same coin as high moderation will bring more thermal neutrons to the pool while high absorption will take more from the pool. Therefore we can treat moderation as just anti-absorbtion and multiply all the important factors to get the final `SomeFactors` constant.

Control Rods

In this simple 1-D model of the control rods position can be used to modify our `SomeFactors` directly:

```
local SomeFactors
local RodsPulled

SomeFactors = 2 * RodsPulled / 100
```

`RodsPulled` is just percent of rods pulled from the core, hence it is divided by 100. The 2 is just an example which will give critical state at 50% of rods pulled. You can already implement this and start playing with it to get the feel of how reactor power changes. If you want to calculate reactor period, the formula is following:

```
local ReactorPeriod
local TimeStep -- number of seconds

ReactorPeriod = 1 / math.log( NumberOfNeutrons
                  / oldNumberOfNeutron ) * 1 / TimeStep
```

You probably don't want to make your rods 100% effective like here, so you might want to try a different formula, like for example:

```
SomeFactors = 2 * ( 0.2 + RodsPulled * 0.8 / 100 )
```

In that case even with all rods in the core, there will still be some neutrons in the core. It may become handy later on, when we will build a 2-D core, as in real life energy is distributed evenly in the core and one rod can't totally kill reactivity in that part.

Density

We will deal with the calculation of water density later, but for now we can already put it in the equation. Water density is the main factor that changes how moderation works. When water density drops, the moderation worsens, so less neutrons are available in the pool, therefore we can treat it similar to absorption. As water density is about $1000\frac{\text{kg}}{m^3}$, the simplest approach is to divide water density by 1000 and put it as another factor:

```
local SomeFactors
local RodsPulled
local WaterDensity = 1000

SomeFactors = 2 * ( 0.2 + RodsPulled * 0.8 / 100 ) * WaterDensity / 1000
```

Xenon

Xenon is an important factor in reactor operations although it's not crucial to the experience, so you can easily deal without it. If you want to implement xenon poisoning I suggest this simple xenon/iodine scheme:

- iodine builds up proportionally to reactor power
- iodine decays with time into xenon

- xenon decays with time
- xenon decays proportionally to reactor power

```
local iodine
local xenon
local iodineDecay
local ScalingFactor = 0.00025

iodine += NumberOfNeutrons / MaxNeutrons * ScalingFactor
iodineDecay = iodine * ScalingFactor
iodine -= iodineDecay

xenon += iodineDecay
xenon -= 2 / 3 * xenon * ScalingFactor * TimeFactor
xenon -= xenon * NumberOfNeutrons / MaxNeutrons * ScalingFactor
```

`ScalingFactor` needs to be tuned to how quickly you want the effects to take place and to your timestep. In real life the effects are in scale of hours but you may decide to make it quicker.

Fuel

Fuel availability is another factor that affects reactivity. It's just a single number denoting the fuel percentage in the given rod. If you want you can implement time/power related depletion of fuel although this will affect reactivity and rods will have to be constantly pulled to account for that. Therefore RBWR uses just fixed amount which is randomized at the start. I don't think it makes sense to simulate a process which takes months in real life so this fuel depletion doesn't affect day-to-day operations.

```
local FuelRemaining -- in percent

SomeFactors = 2 * ( 0.2+RodsPulled * 0.8 / 100 ) * WaterDensity / 1000
                * FuelRemaining * ( 1 - xenon / 4.5 )
```

Xenon must be reduced from reactivity and 4.5 comes just from fine tuning, you can use a different number.

Steam voids

Unfortunately I don't have any good idea how to simulate steam voids. I suggest you make them just proportional to total power. If you want to make it a bit more realistic, you can make them inversely proportional to pressure too which would allow for voide collapse mechanics. RBWR classic reactor model just uses this formula:

```
local VoidCoefficient = 1 - 0.3 * NumberOfNeutrons / MaxNeutrons

SomeFactors = 2 * ( 0.2 + RodsPulled * 0.8 / 100) * WaterDensity / 1000
                * FuelRemaining * ( 1 - xenon / 4.5 ) * VoidCoefficient
```

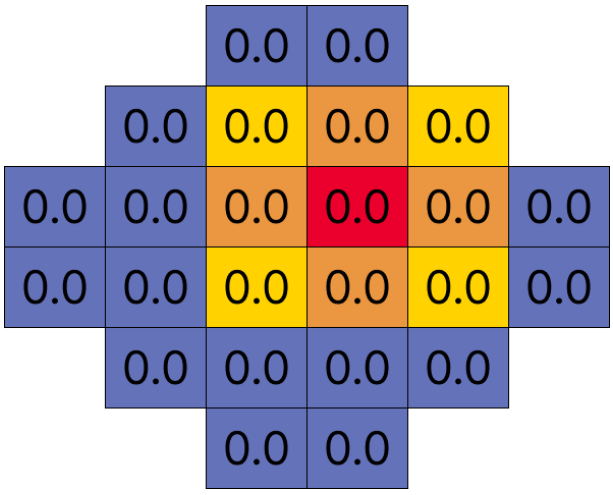
BWR reactor have recirculation pumps. They should reduce the effect `VoidCoefficient` (therefore at full recirculation it should equal 1). This will naturally lead to raise in reactivity.

2-D core

You will probably not be satisfied with the simple 1D core and want to implement a 2 dimensional core consisting of several rods (RBWR uses 24, real reactors have around 200). In real life control rods and fuel rods have their own channels but I would argue to simplify things, that a single channel serves as both a fuel bundle and a control rod. Your goal is to build a 2-D array where every cell gets its value updated by your formula. You will need similar arrays for all the factors like fuel, xenon, rods, etc. This way your core consists of several independent smaller cores. The last step would be:

Neutron transfer

In your current setup every rod is independent, but in real life rods affect their neighbors because neutrons can travel between them. Therefore hot rods should heat up their neighbors while cold rods should cool them down. In general this problem is complex and would require some kind of ray tracing of the neutrons but I suggest a simple procedure here, where we transfer neutrons only to the neighbors and if you want you can iterate it many times per step.



Let's suppose we are looping over all cells and we are now at the red one. What we want to do is to cut a fraction of the neutrons from the cell, say 50%, and prepare a temporary matrix in which we will put the transferred neutrons. By geometry you can assume that $\frac{1}{6}$ of the neutrons would go to each orange cell and $\frac{1}{12}$ to the yellow ones. This way $4 \cdot \frac{1}{6} + 4 \cdot \frac{1}{12} = 1$ you exactly distribute all of them. You can take different proportions but be sure that it equals one, so no neutrons are missing nor any are artificially created. Just as an example if we had 1000 neutrons in the red cell, we cut 500 leaving 500, and distribute them proportionally $\frac{500}{6} = 83$ to the orange cells and $\frac{500}{12} = 42$ to the yellow ones. Once the procedure is complete for all the cells, we now add up the temporary array to our current

neutron array (after cuts) and this is our new `oldNumberOfNeutrons` array to which we will apply our reactivity formula for calculation of the next step. What about cells at the edges? You can assume that the neutrons are just missing to the outside but you can also apply some neutron reflection there. Just take a fraction of neutrons that would be normally lost and put them back in your cell with some reflection effectiveness say 60%. Depending on what percentages you chose for transfer and reflector you will get higher or lower contrast between the cells. You will need to tune this up to get the right feel that you expect. You can repeat the procedure of neutron transfer many times per step. If you do it only once, we can assume that each rod only affects its neighbors but if you do it several times, then you simulate longer neutron distances. After each iteration you should apply your absorbtion/moderation coefficient, but calculate new number of neutrons only once at the end of the process.

Water heating

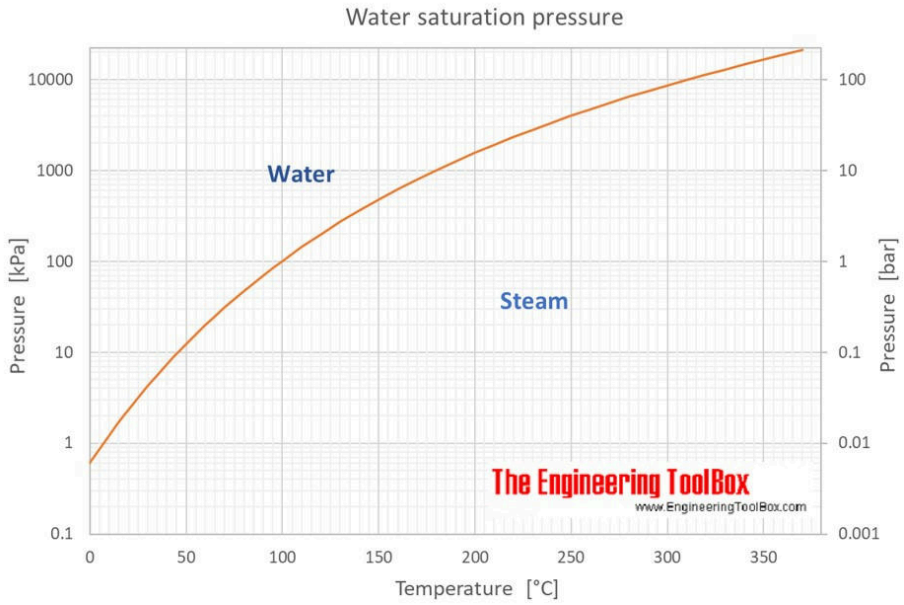
Water heating has quite simple physics. Specific heat of water is around $4200\frac{\frac{J}{kg}}{°C}$ which means that it requires 4200 joules to heat up 1 kg of water by 1 degree. If you want you can do it in a fully physical way but I would just suggest to take the tuning up approach. In other words let's answer the question: "How quickly do you want your reactor to heat up on, say, 10% of reactor power?" For a BWR you will need to heat it up from around 20 degrees up to 280, but above 100 degrees it will take longer, as there will also be boiling. I would argue that this should happen in matter of minutes because longer time will just make the game boring. Therefore let's take following constants:

```
local WaterTemperature = 20
local HeatPerPower = 10 -- degrees per timestep for 100% reactor power
WaterTemperature += NumberOfNeutrons/MaxNeutrons * HeatPerPower
```

You probably don't want to put water temperature in an array because you'd have to implement some kind of mixing which could be tricky. In RBWR water temperature is just one parameter for the whole core. Therefore `NumberOfNeutrons` in above formula is just an average for all cells. Also please note here that we do not input water mass (or volume) anywhere here. This is because I would assume that water level won't change a lot in the reactor (comparing to how much water there is) but if you want a more sophisticated simulation, you should divide temperature gain by water amount in the reactor (more water = slower heating). Water amount will be dealt with later.

Water boiling

Water boiling is a bit tricky because it is important to understand saturation temperature which is the temperature at which water boils. It is 100℃ under atmospheric pressure, but this will raise as the pressure raises. Therefore we will need this saturation temperature as a function of pressure to continue. You can find a graph for saturation temperature like this one:



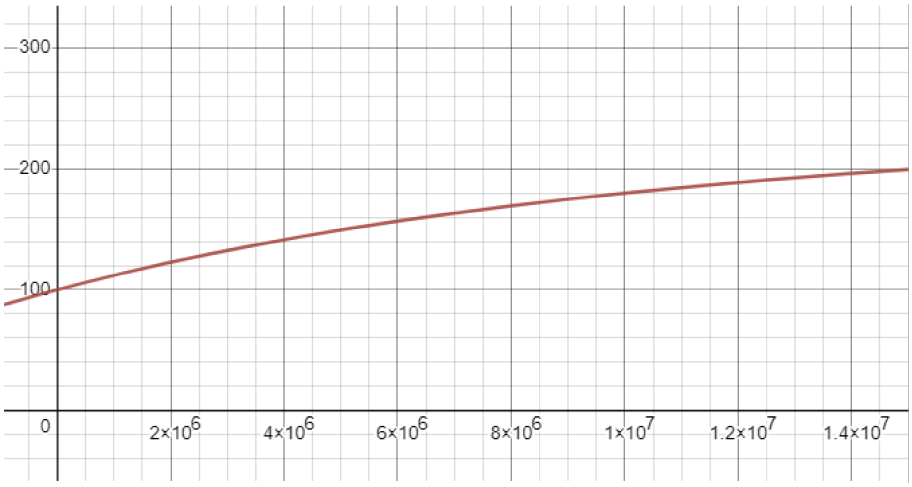
Pressure is usually rescaled so 0 means atmospheric, therefore with no steam at 0 pressure we have 100 degrees of saturation temperature. When the pressure rises saturation temperature will rise too. This is done by using a curve fitter to aproximate the above plot. You can find a curve fitter and make your own fit, or use mine:

```
function CalculateBoilingPoint(pressure) -- pressure in Pa
  pressure*=10

  if pressure/10000000>20 then -- above this physics breaks
    pressure=20*10000000
  end

  local temperature =   1.0002646407033929 * 10^2
                      + 1.2485512687579338 * 10^( -05 ) * pressure
                      - 6.0865396119409739 * 10^( -13 ) * pressure^2
                      + 1.8961630152963698 * 10^( -20 ) * pressure^3
                      - 3.4939669493500672 * 10^( -28 ) * pressure^4
                      + 3.8256436415169860 * 10^( -36 ) * pressure^5
                      - 2.4385640912095220 * 10^( -44 ) * pressure^6
                      + 8.3341353038170792 * 10^( -53 ) * pressure^7
                      - 1.1778965239174813 * 10^( -61 ) * pressure^8
  return temperature -- temperature in degrees celsius
end
```

Which looks like this:



When water temperature reaches the saturation temperature it will start to boil. Again you can use real the physical values here for specific boiling heat or tune them with your own constants based on how quickly you want pressure to raise. What happens when water temperature raises above saturation temperature? There is a surplus of temperature (or energy) which will cause boiling. Boiling will reduce the amount of water in the reactor and will increase amount of steam. Next we need to calculate pressure. Pressure is proportional to the amount of steam and temperature divided by volume. I would assume that volume is constant but you can modify volume available for the steam based on water level (lower level gives more volume hence lower pressure).

```
local WaterAmount
local SteamAmount
local WaterTemperature
local TemperatureSurplus
local SaturationTemperature
local BoilingCoefficient -- tune this
local AmountBoiled
local Pressure
local SteamPressureCoefficient -- tune this

if WaterTemperature > SaturationTemperature then
  TemperatureSurplus = WaterTemperature - SaturationTemperature
  AmountBoiled = TemperatureSurplus * BoilingCoefficient

  WaterTemperature = SaturationTemperature

  WaterAmount -= AmountBoiled
  SteamAmount += AmountBoiled

  Pressure = SteamAmount * SteamPressureCoefficient *
             ( WaterTemperature + 273 ) / 373

  SaturationTemperature = CalculateBoilingPoint(Pressure) -- update for next
                                                          -- timestep
end
```

The addition of 273 is to translate temperature into kelvins and division by 373 makes it equal to one at 100 degrees. As you can see water temperature can't be higher than current saturation temperature and the saturation temperature for the next step is calculated once we have pressure (from amount of steam and temperature). At this stage, you should also calculate water level. Water level will depend on amount of water divided by water density. Water density is also needed as an input for the reactivity calculations so you definitely need to calculate it at some point. Again you can use a polynomial fit to a known physical function or use my fit:

```
function CalculateWaterDensity(temperature) -- temperature in celsius
  local density = -0.000004467711 * temperature^3
                  -0.000560288485 * temperature^2
                  -0.429148844451 * temperature
                  +1010.035413387815
  return density
end

WaterDensity = CalculateWaterDensity(WaterTemperature)

local WaterLevel
local AmountToLevelCoefficient -- tune this

WaterLevel = WaterAmount/WaterDensity * AmountToLevelCoefficient
```

Final steps to finish of the reactor would be to implement feedwater flow and bypass/ turbine outflow.

Feedwater

Water is fed into the reactor by feedwater pumps. You will need to design some controls for the pumps. In RBWR pumps are quite simple, they only have setpoints, RPM and flows. RPM can have some time lag before they reach desired setpoint and same goes to the flows. This kind of pump lag can be easily simulated:

```
local RPMSetpoint -- set by user
local PumpRPM
local PumpFlow
local PumpLag -- tune this
local RPMToFlowCoefficient -- tune this

PumpRPM = PumpRPM + ( RPMSetpoint - PumpRPM ) / PumpLag
PumpFlow = PumpRPM * RPMToFlowCoefficient
```

With this setup the pump will gain or lose RPM proportionally to the distance of the given setpoint. When pump trips it is good to put the setpoint at negative, so the pump stops quicker, but of course RPM will have to be clamped to not go below 0. This flow has no physical equivilant. It uses the same scaling as water amount. When you want to display the values for the users in kg/s or similar, you will have to multiply them by some constant to make it realistic (for example to tune it so it matches the maximum flow of a real feedwater pump). You can also make things a bit more realistic adding some flow coefficient based on pressure as it is harder to pump water against higher pressures, it's up to you.

Adding water to the core will obviously add to the amount but also will change temperature due to mixing. Feedwater will have part of its temperature dependent on preheaters and deaerator (if existent).

```
local MixingParameter = 0.1 -- tune this
local FeedwaterTemperature

WaterTemperature = ( WaterTemperature + PumpFlow * FeedwaterTemperature *
MixingParameter ) / ( 1 + PumpFlow * MixingParameter )
WaterAmount += PumpFlow
```

Turbine/Bypass valves

Amount of steam that goes away through bypass or turbine valve will depend on pressure. Therefore:

```
local TurbineSetting -- set by user in %
local TurbineSteamCoefficient -- tune this up
local TurbineSteam
TurbineSteam = TurbineSetting * TurbineSteamCoefficient * Pressure
SteamAmount -= TurbineSteam
```

Be sure to clamp steam amount so it never goes negative. The coefficient should be tuned in a way that at normal operating pressure (7.1 MPa) and temperature (278 degrees) pressure can be maintained with turbine valve opened close to maximum (80-90%). You probably will want to make bypass weaker than turbine valve but it works pretty much similar. Also when testing, note the `TurbineSteam` at 100% of reactor power, as this will be your guide on how to configure the turbine.

You are pretty much done.

Turbine

The Turbine works in two modes. One is when it is not synced, therefore it will spin up freely depending on steam flow and pressure, the other one where generator is synced and constant RPM is enforced. Status of the main generator breaker (the sync switch) will determine the mode. Turbine output depends mostly on steam flow and pressure, although you might want to implement some function that takes into account the efficiency, which would be higher at optimal pressure. In real life the turbine power output is very complex but this simplification works quite well.

Free spin mode

Depending on where the reactor is located, the network can be either 50 Hz or 60 Hz. This will determine RPM for synchronization either 3000 or 1500 for 50 Hz network or 3600 and 1800 for 60 Hz network. RBWR uses 3600 as synchronization RPM. Next you should decide at what reactor power the turbine should sync. RBWR uses 10% but in real life probably around 5% power would be a good call. Once decided it would be the easiest to run the reactor at the desired sync power, stabilize it at the desired pressure and adjusting the turbine valve to keep constant pressure and check actual steam flow. This is the turbine formula:

```
local RPM
local RPMGoal
local SteamRPMCoefficient -- will calculate in a moment
local TurbineSteam -- outflow based on turbine valve setting
local TurbineLag -- tune this

RPMGoal = TurbineSteam * Pressure * SteamRPMCoefficient
RPM = RPM + (RPMGoal - RPM)/TurbineLag
```

As you can see this uses the same lag idea as the pumps, as obviously you don't want turbine to spin up immediately when applying more steam flow. When turbine is synchronized it will maintain constant desired RPM, while generated power will depend on steam flow and pressure. It is important that the steam flow that allows the turbine to match your sync RPM gives exactly 0 MW of power, while at your 100% reactor power you want to produce the maximum rated electrical power you chose. Therefore you need to find the `SteamRPMCoefficient` first. We can calculate it if we know exact steam flow for the sync conditions:

```
local SyncTurbineSteam -- exact steam flow at your sync power and pressure
local SyncRPM -- RPM at which you synchronize
SyncRPM = SyncTurbineSteam * SyncPressure * SteamRPMCoefficient
SteamRPMCoefficient = SyncRPM / ( SyncPressure * SyncTurbineSteam )
```

Synchronized mode

Next assuming that you know what is the `TurbineSteam` for maximum reactor power, you can create the second formula:

```
local SteamToMW -- will calculate this in a moment
GeneratorLoad = (TurbineSteam/SyncTurbineSteam - 1) * SteamToMW
```

By transforming this equation we get:

```
local MaxGeneratorLoad -- generator load for MaxSteamFlow ie. 1000 MW
local MaxTurbineSteam -- steam flow for 100% reactor power at constant
pressure
MaxGeneratorLoad = (MaxTurbineSteam/SyncTurbineSteam - 1) * SteamToMW
SteamToMW = MaxGeneratorLoad /(MaxTurbineSteam/SyncTurbineSteam - 1)
```

Finally we get formula for the generator load:

```
local RPM = 3600 -- or any value you chose
local TurbineSteam -- outflow based on turbine valve setting
local GeneratorLoad
local SteamToMW -- from above
GeneratorLoad = (TurbineSteam/SyncTurbineSteam - 1) * SteamToMW
```

Just to summarize. In order to tune up your turbine you need to have a turbine valve which can operate up to full reactor power at a desired constant pressure. Then run your simulation and check the steam flow in 2 conditions. One is the synchronization RPM you chose, second one is the maximum reactor rated power for an electrical output that you chose. Having these two steam flows, you can calculate the two important constants: `SteamRPMCoefficient` for a free spin and `SteamToMW` for a synced state.

Synchronization

In order to synchronize the generator, several conditions should be met. RPM must be close to sync RPM, acceleration of the turbine must be close to 0 and phase must also be close to 0. Phase can be calculated in the following way:

```
local phase = math.fmod(phase + PhaseCoefficient * (RPM - SyncRPM), SyncRPM)
```

`PhaseCoefficient` will depend on the timestep for the turbine. In RBWR where turbine is calculated 20 times per second, the coefficient equals 3. Such calculated phase is actually measured in the units of SyncRPM, if you want classical degrees you need a new variable where you divide it by SyncRPM and multiply by 360. Also phase can be positive or negative, so you might want to add 360 to the negative values. Resulting angle should directly drive your Synchroscope, with phase = 0 at the top.

```
local angle = phase/SyncRPM * 360
if angle < 0 then angle += 360 end
```

Condenser

The Condenser is used to condensate used steam back into water. During this process the steam becomes a lot more dense therefore creating a vacuum in the Condenser. This will suck out steam from the turbine increasing efficiency. You can simulate this process by adding condenser vacuum to main steamline pressure before applying turbine formulas although assuming that condenser pressure doesn't change a lot during operations I find it pointless.

Unfortunately the condenser code in RBWR is not perfect and it requires a rework so I don't want to show you the code itself, instead I will give you an idea on how it should work.

There should be two components in the condenser – steam and air. Steam is just added the normal way through bypass and turbine valve, then you can calculate steam pressure for it the same way as you did for the reactor (just multiply steam and air amount by some factor). Air just fills the condenser initially and it's pressure should be equal to atmospheric. Adding steam will increase pressure. The Condenser should have devices that reduce pressure like Condenser Air Removal and Steam Jet Air Ejectors. When enabled you can assume they remove air. There should be also a hole in the condenser which adds air proportionally to the vacuum which will increase pressure back to atmospheric during shutdown. Finally there is circulation flow which will change steam into water, filling the hotwell.

Now let's assume that you filled your condenser with steam and therefore ejected all air. When you enable the recirculation circuit, steam will change into water proportionally to the recirculation flow, therefore the steam amount will drop reducing pressure. When you provide more steam, the pressure will rise again, but if you can provide the right recirculation flow to condense the exact same amount of steam that is coming in, in doing so the steam amount will remain constant, so will the pressure.

The Typical operating pressure for a condenser is 70 mbar, so you need only about 7% of initial steam amount in the condenser during operations.

Automatic controls

RBWR uses automatic control for all systems based on a very simple scheme. It is based on calculating distance from the desired setpoint and a rate of change.

Let's assume we want to control reactor water level by changing the amount of water inflow.

```
local WaterLevel
local oldWaterLevel – timestep before
local WaterLevelSetpoint – desired level set by the user
local RangeCoefficient – tune this
local RateCoefficient – tune this
local FWPumpSetopint – pump setting adjusted by autocontrol
local Range
local Rate

Range = (WaterLevelSetpoint – WaterLevel) * RangeCoefficient
Rate = (WaterLevel – oldWaterLevel) * RateCoefficient
FWPumpSetpoint += (Range - Rate)
```

Let's say that water level is 0 units and the setpoint is at 5. Range would equal to 5. Autocontrol is trying to open the valve by 5 per step to add more water. What if water level is at 4? Then range would be 1, so autocontrol would still try to open the valve but slower (only by 1). At level of 5, the valve would stay would stop moving.

Now for the second parameter which is rate. If water level was at 0 and it didn't change at all, autocontrol wouldn't touch the valve, so only range coefficient would keep opening it. Soon water level would start to raise, so the rate would become positive. Therefore the rate component would try to close the valve working against the range component. The quicker the change, the quicker the rate component would try to close the valve, while nearing to the setpoint the input from range component would be smaller. Therefore at some stage before reaching the setpoint, the rate components value would be higher than range component and as total it would start closing the valve. Hopefully both components would equal 0 at the right water level. So it's the balance between range and rate that drives the pump valve.

The closer we are to the setpoint, the more importnt the rate component becomes, the farther we are the more important the range one is. You can use this scheme for any type of autocontrol you have like reactor power, condenser pressure, turbine RPM etc. It's just a matter of tuning the parameters. If the autocontrol tends to overshoot the setpoint, try giving higher coefficient to the rate, so it reacts quicker. If autocontrol struggles to reach the setpoint, give lower coefficient to the rate. If autocontrol reacts too slowly, increase both coefficients by the same factor etc. Of course many of the devices have some speed limits, for example you can't pull the rods very quickly, so you need to clamp the result into acceptable limits.

Often you will want to add additional factors to the formula. For example autocontrol for rods should check the period and if it's too low, shouldn't pull the rods even if the distance is so great that it would like to (as user can preselect 100% setpoint on 1% reactor power).

More advanced stuff

3D Core

You can decide to build a fully three dimensional core. This is how RBWR is modeled now. However probably you don't want it, as it really has a minimal impact on operations and requires much more computational power. It makes sense only if your core is small. For a 3-d core you need to divide your rods into vertical slices and reactivity formula will have to be applied to each cell. RBWR uses 10 slices for a total of 24*10=240 cells. You'd also need to add vertical, so up and down, neutron transfer. Rods are a bit tricky in this scheme. Let's say your rods are pulled 16% from the core. Assuming it's a BWR, so rods are pulled down, you will have:

- 0% rods in the top cell
- 40% in the second cell from the top
- 100% in the 8 bottom cells

This is why I mentioned before that a fully inserted rod shouldn't kill reactivity to zero. You will have to find proper values for the rod absorbtion so that there is some reactivity even with a fully inserted rod. Note here that as many cells will have 0% rods inside, they can't go critical by themselves, so you will need plenty of neutron transfer (the more neutron transfer the higher the critical mass for your reactor becomes). You will have to do a lot of experiments to tune this up and get the right feel of the reactor. If you are building a RBMK type reactor you can even simulate graphite displacer. If the rods are partially inserted into the core, there would be a part with higher absorption, a part with no absorption and a part with higher moderation (where the displacer is). It is done this way in RBWR when RBMK model is selected.

Steam voids also need a rework, as voids generally travel up you want to add a rising void effect for each cell so its affected by all cells bellow it. This way you should have a natural pattern, where rods block reactivity at the bottom (if not fully pulled), while voids block it at the top, therefore the middle of the core is the hottest as in real life. Exploring 3-d models even more, you could introduce a vertical water temperature gradient in the core, which would be reduced by recirculation flow, simulating better water mixing. There are a lot of possibilities, but be aware that all that work will really have a minimal impact on reactor operations, and 2-d model is pretty good itself.

Islanding mode

Islanding mode is defined as operations where the turbine generator is not synchronized with the network, but is powering the main busses. In that case obviously the turbine runs in free spin mode, as there is no network that would keep it at constant RPM. However as

the busses and equipment are designed for the network frequency, user should still keep it around your desired sync RPM and when far away from it the breaker should open.

For islanding mode you will use standard free spin turbine formula, although you need to modify it by a load on the generator. Load induces electrical 'drag' and requires more steam to keep the turbine on given RPM. On the other hand when you reduce the load, turbine will have a tendency to spin up and steam flow should be reduced. This comes from natural conservation of energy. Let's have a look at our free spin formula:

```
RPMGoal = TurbineSteam * Pressure * SteamRPMCoefficient
RPM = RPM + ( RPMGoal - RPM ) / TurbineLag
```

As we already know how many MW we get from steam by `SteamToMW` variable, we can directly implement it here:

```
local LoadMW -- power required by all equipement connected to the generator

RPMGoal = ( TurbineSteam - LoadMW * SteamToMW ) * Pressure *
SteamRPMCoefficient
RPM = RPM + ( RPMGoal - RPM )/TurbineLag
```

Remember that whenever offsite power is reconnected, turbine generator will have to be resynced back to the offsite, therefore generator breaker will have to open.

Conclusions

Presented topics should give you some basic ideas on how to build a simple nuclear power plant model. This definitely doesn't provide everything you need but at least you have a place to start. You don't really need to copy it one to one. I hope that you can come up with your ideas which might be much better than in RBWR. If so we will be glad to see your solutions and possibly use them to improve our own experiences.

Still remember that your game will need a lot more than just physics. You will need to design a gameplay that will keep people playing. You need to build environment that is easy to learn yet still challenging to master. I believe that you should design your experience to be as simple as possible but without sacraficing the realism you are aiming at. You may have an impression that RBWR is too simple, but definitely it works and keeps people playing. If you try to implement some complex physics always ask yourself a question: "Will this have any real impact on the gameplay?". Too much time spent on a meaningless code is not only a loss for the community, which will wait for the experience for a longer time, but also can jeopardize your whole project as you can get burnt out before you manage to finish it. Physics is important but it's still just one element of a successful experience. Don't waste all your effort on it. You can always improve it post release.

Be advised that there might be some errors in the presented parts of the code. If you find any please report them on our Discord server, so we can improve the document for further releases.